Modeling and Simulation of Internet of Things Infrastructures for Cyber-Physical Energy Systems

Dissertation

Vom Fachbereich Informatik der
Rheinland-Pfälzischen Technischen Universität Kaiserslautern-Landau
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

von

Johannes Koch

Datum der wissenschaftlichen Aussprache	28.02.2024
Dekan	Prof. Dr. Christoph Garth
Vorsitzende der Promotionskommission	Prof. Dr. Annette Middelkoop-Bieniusa
Gutachter	Prof. Dr. Christoph Grimm
	Prof. Dr. Peter Palensky

DE-386

Abstract

This dissertation presents a novel approach to the model-based development and simulation-based validation of Internet of Things (IoT) infrastructures within the context of Cyber-Physical Energy Systems (CPES). CPES represents an evolution in energy management, seamlessly blending physical and cyber components for efficient, secure, and dependable energy distribution. However, the intricate interplay of these components demands innovative modeling and simulation strategies. The work begins by establishing a robust foundation, exploring essential background elements such as requirements engineering, model-based systems engineering, digitalization approaches, and the intricacies of IoT platforms. It introduces the novel concept of homomorphic encryption, a critical enabler for securing IoT data within CPES. In the exploration of the state of the art, the dissertation delves into the multifaceted landscape of IoT simulation, emphasizing the significance of versatility, community support, scalability, and synchronization. The core contribution emerges in the chapter on simulating IoT networks. It introduces a sophisticated framework that encompasses hardware-in-the-loop, software-in-the-loop, and human-in-the-loop simulation. This innovative framework extends the boundaries of conventional simulation, enabling holistic evaluations of IoT systems. A practical case study on smart energy usage showcases the application of the framework. Detailed SysML models, including requirements, package diagrams, block definition diagrams, internal block diagrams, state machine diagrams, and activity diagrams, are meticulously examined. The performance evaluation encompasses diverse aspects, from hardware and software validation to human interaction. In conclusion, this dissertation represents a significant leap forward in the integration of IoT infrastructures within CPES. Its contributions extend from a comprehensive understanding of foundational elements to the practical implementation of a holistic simulation framework. This work not only addresses the current challenges but also outlines a path for future research, shaping the landscape of IoT integration within the dynamic realm of CPES. It offers invaluable insights for researchers, engineers, and stakeholders working towards resilient, secure, and energy-efficient infrastructures.

Kurzfassung

Die vorliegende Dissertation präsentiert einen innovativen Ansatz für die Modellierung und Simulation von Internet der Dinge (IoT)-Infrastrukturen in Cyber-Physische Energiesysteme (CPES). CPES verbinden physische und cyberbasierte Komponenten zur effizienten Energieverteilung und sind ein zentraler Baustein für die Energieversorgung der Zukunft. Die Arbeit beginnt mit einer gründlichen Untersuchung des Anforderungsmanagements und des modellbasierten Systems Engineering, um eine solide Grundlage zu schaffen. Sie hebt die wachsende Bedeutung von IoT-Simulationen hervor und stellt die Herausforderungen bei der Modellierung und Simulation von komplexen und skalierbaren IoT-Systemen heraus. Das Forschungsprojekt entwickelt ein neuartiges Framework, das Hardware-in-the-Loop-, Software-in-the-Loop- und Mensch-in-the-Loop-Simulationen ermöglicht. Dieses Framework erlaubt eine umfassende Bewertung von IoT-Systemen, einschließlich Hardwarekomponenten, Softwareintegration und menschlicher Interaktion. Ein zentrales Highlight dieser Arbeit ist eine tiefgehende Fallstudie zur intelligenten Energieverwendung. Diese Fallstudie umfasst umfangreiche SysML-Modelle, die detailliert evaluiert werden. Dabei werden Leistungstests für Hardware und Software durchgeführt und menschliche Interaktionen in die Simulation integriert. Die Dissertation trägt nicht nur zur nahtlosen Integration von IoT-Infrastrukturen in CPES bei, sondern bietet auch wertvolle Erkenntnisse für Forscher und Ingenieure, die an sicheren und energieeffizienten Infrastrukturen arbeiten. Die entwickelten Modelle und das Framework haben das Potenzial, die Entwicklung und Integration von IoT-Systemen erheblich zu verbessern und die Entstehung der nächsten Generation von CPES zu fördern. Dieser Forschungsbeitrag ebnet den Weg für zukünftige Innovationen im Bereich der Energieversorgung und des Internet der Dinge.

Acknowledgements

Completing this dissertation would not have been possible without the support and encouragement of many individuals and institutions, and I would like to take this opportunity to express my gratitude to them.

First and foremost, I extend my deepest appreciation to my academic advisors, Prof. Dr. Christoph Grimm and Prof. Dr. Peter Palensky. Their guidance, expertise, and unwavering support have been the cornerstone of this research. Their mentorship has not only shaped this work but has also been instrumental in my growth as a researcher.

In the realm of academia and the pursuit of knowledge, where time is often consumed by research and scholarly endeavors, there exists an unwavering pillar of love and support that remains steadfast. To my beloved wife Anne, your unwavering love, boundless patience, and ceaseless encouragement have been the guiding stars illuminating this arduous journey. Your belief in me, even during the most challenging moments, has been a wellspring of inspiration. Your sacrifices and understanding, as I delved into the depths of research and writing, are immeasurable. This dissertation is not just a culmination of academic endeavors but a testament to the strength of our partnership. I express my deepest gratitude for your love, which has been the foundation upon which this achievement rests.

To my family and friends, I owe a debt of gratitude that words cannot adequately express. Your belief in me, your patience, and your unending encouragement during the challenges of this academic journey have been my driving force. Your unwavering support has meant more to me than I can convey.

I would also like to acknowledge my colleagues, research peers and students who have been with me throughout this academic voyage over the years. Your collaborative spirit, insightful discussions, and constructive feedback have enriched this research immensely.

My sincere appreciation goes to the whole RPTU for its unwavering support and the resources it provided. Their commitment to fostering academic excellence has been pivotal.

Lastly, I want to extend my thanks to anyone not mentioned here who has provided assistance, shared insights, or offered encouragement during this research journey.

The completion of this dissertation has been a collective effort, and I am deeply thankful to all those who have been part of this academic adventure. Your contributions have not only enriched the quality of this work but have also played a significant role in my personal and academic growth as a researcher.

28.02.2024, Johannes Koch

If I have seen further
it is by standing on the shoulders of giants.

ISAAC NEWTON

Contents

List of Figures

Acronyms

API	Application Programming Interface
CPS	Cyber-Physical System
CS	Computer science
DARPA	Defense Advanced Research Projects Agency of the United States of America
dev	Development
DEVS	Discrete Event System
DNS	Domain Name System
DSM	Demand Side Management
EV	Electric Vehicle
FBR	Fachbereichsrat
FSM	Finite-State Machine
GPS	Global Positioning System
H2M	Human-to-Machine
HCI	Human Computer Interaction
HiL	Hardware-in-the-Loop
HTTP	Hypertext Transfer Protocol
I/O	Input / Output
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IoT	Internet of Things
IoT-EPI	IoT European Platforms Initiative
IP	Internet Protocol
IPv4	IP Version 4
ISO	International Organization for Standardization
IT	Information Technology
ITU	International Telecommunication Union
JSON	JavaScript Object Notation

LAN	Local Area Network
M2M	Machine-to-Machine
MBSDK	Mercedes-Benz Mobile Software Development Kit
MVC	Model View Controller
MVVM	Model View View-Model
NASA	National Aeronautics and Space Administration
NED	Network Description
NFC	Near Field Communication
NP	Non-deterministic polynomial time (complexity class)
oid	Object-Identification
OS	Operating System
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
pid	Property-Identification
PLIST	Property List
REST	Representational State Transfer
RFC	Request for Comments
RPTU	Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau
SOTA	State-of-the-Art
TCP	Transmission Control Protocol
TUK	Technical University of Kaiserslautern
URI	Universal Resource Identifier
UX	User-Experience
VLSI	Very Large Scale Integration
W3C	World Wide Web Consortium
WoT	Web of Things
XML	Extensible Markup Language

Chapter 1

Introduction

Contents

1.1. Starting Situation

The realm of Cyber-Physical Systems design and simulation is confronted with a dynamic landscape characterized by an array of challenges stemming from the rapid evolution of products, their utilization, and development methodologies. This ongoing evolution, spurred by the ever-changing demands of consumers and market conditions, engenders a considerable augmentation in the intricacy of products. Such intricacy is manifest in the proliferation of components, product variants, and interconnections within these systems. These intricate products are not confined to the domain of the Internet of Things (IoT) but radiate into various sectors, including but not limited to automotive and electric grid applications.

Additionally, the deployment of Internet of Things (IoT) networks has witnessed exponential growth in recent times, as portrayed in Figure 1.1. These IoT networks span various domains, such as smart cities, energy, eHealth, and transportation, often operating as isolated entities within the broader IoT landscape [2] (see Figure 1.2). The challenge emerges in seamlessly connecting these devices and isolated networks to establish a coherent network that

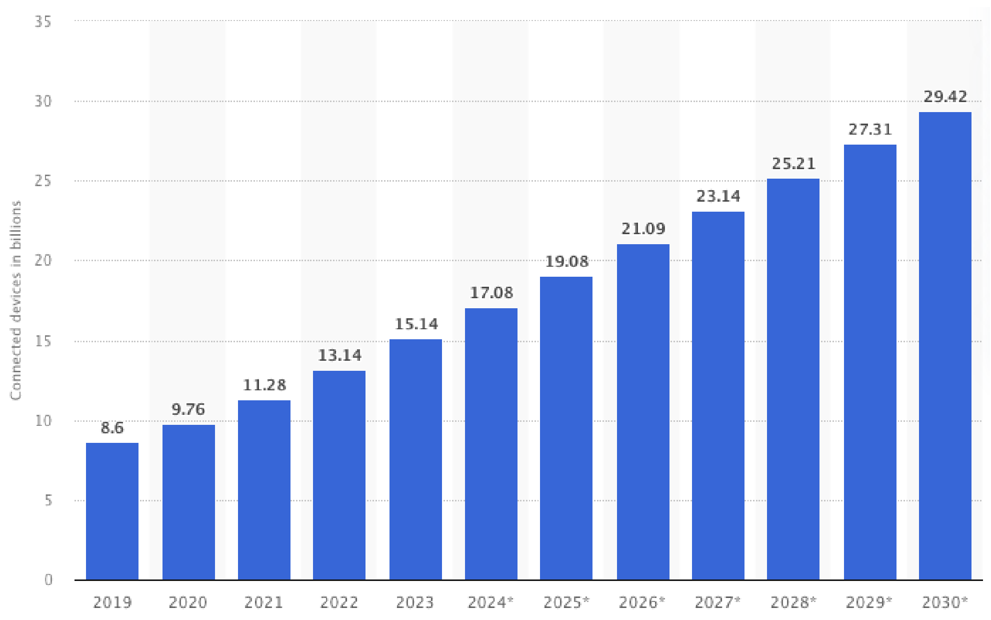


Figure 1.1.: *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2023, with forecasts from 2024 to 2030[1]*

collectively pursues common objectives. This endeavor extends further into harnessing these technological advancements to construct innovative business models within this context.

A striking demonstration of the profound influence of Cyber-Physical Systems within a specific domain can be found in the industrial sector. Over the past two centuries, the industrial landscape has undergone four pivotal revolutions that have left indelible imprints on its trajectory. The first industrial revolution emerged in the late 18th century with the advent of the steam engine, dramatically revolutionizing manufacturing processes. Subsequently, in the early 20th century, the integration of electrical assembly lines catalyzed the era of mass production, further reshaping the industrial landscape. The third industrial revolution materialized in the early 1970s with the onset of actual automation, marked by the deployment of advanced technologies within manufacturing processes. Presently, we are entrenched within the fourth industrial revolution, where Cyber-Physical Systems and information technology have taken center stage in redefining the industrial domain (see Figure 1.3). The fourth industrial revolution represents a paradigm shift in the way industries operate, harnessing interconnected systems and advanced technologies to spur innovation and enhance the efficiency of manufacturing processes. This paradigm shift aligns with the overarching trajectory of the Cyber-Physical Systems landscape, shaping its course with unprecedented influence and multifaceted implications.

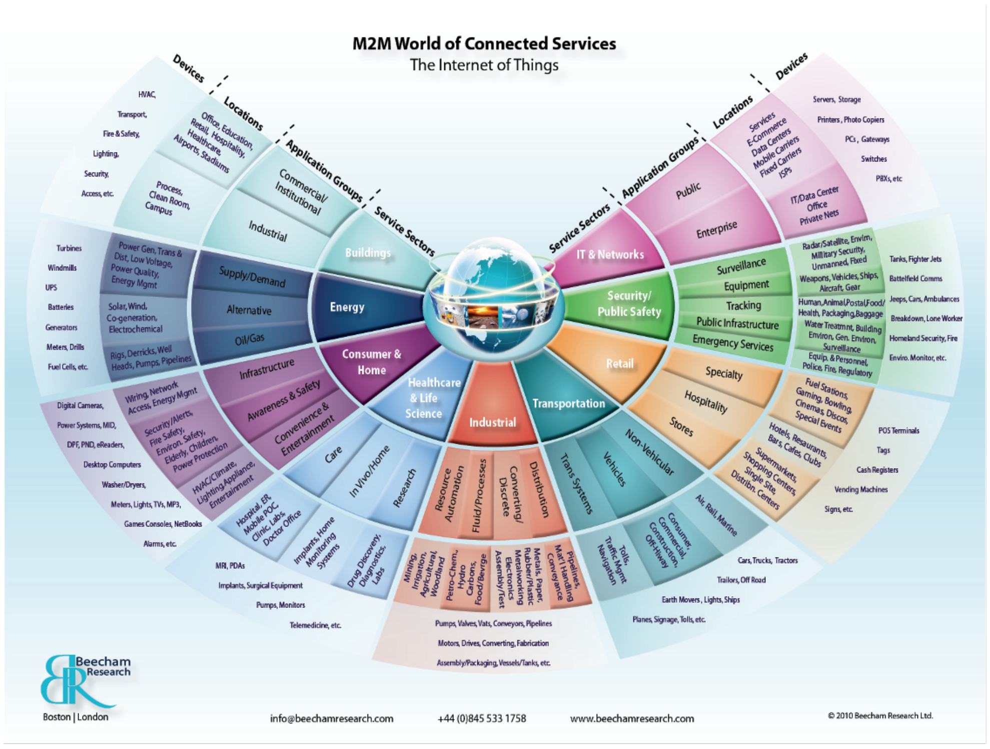


Figure 1.2.: *Connected Services and Devices in the Internet of Things[3]*

1.2. Motivation

The rapid advancement and integration of software and electronics in contemporary goods have given rise to unprecedented complexity in product development. As these products become more sophisticated and interconnected, effectively managing their complexity becomes paramount. Various factors contribute to this complexity, including the proliferation of components, extensive variations, the emergence of intricate systems of systems, and uncertainties surrounding system boundaries and communication partners. In the face of evolving consumer demands, organizations must possess the agility to develop innovative products to maintain their competitiveness in the global marketplace. However, this necessitates a comprehensive understanding of the underlying product complexity and the intricacies of the development processes involved (refer to Figure 1.4).

In the software and electronics industries, where complexity has surged exponentially, there is an urgent need to adopt new development processes and methodologies. These approaches must effectively support the entire lifecycle of a product, encompassing its inception, requirement definition, and eventual disposal and recycling. Moreover, successful collaboration across disciplinary boundaries, transcending the conventional domains of mechanics, electronics, software, and services, is vital for interdisciplinary teams to thrive. Despite the increasing significance of these challenges, there exists a notable research gap in leveraging innovative development methods and concepts to overcome

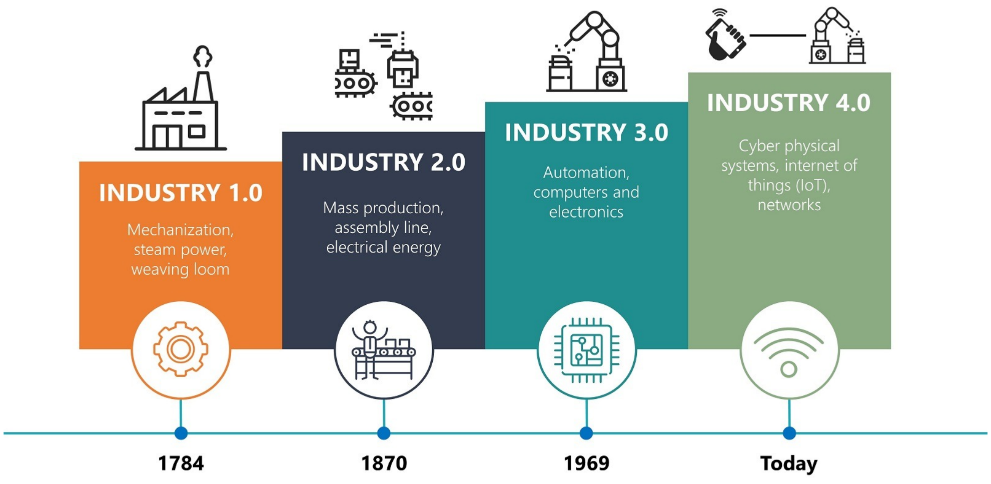

INDUSTRY 1.0
Mechanization, steam power, weaving loom
INDUSTRY 2.0
Mass production, assembly line, electrical energy
INDUSTRY 3.0
Automation, computers and electronics
INDUSTRY 4.0
Cyber physical systems, internet of things (IoT), networks
1784
1870
1969
Today

Figure 1.3.: *The four industrial revolutions in context[4]*

these limitations.

Thus, the primary objective of this dissertation is to address this research gap and contribute to the scientific understanding and practice of managing complexity in contemporary product development. This research endeavors to explore and propose novel approaches that facilitate the construction of robust development processes and innovative concepts, enabling effective complexity management and fostering seamless collaboration across disciplines. Special emphasis will be placed on designing methods and frameworks that streamline the efficient creation of cutting-edge products, accounting for the intricate interplay between software, electronics, and other relevant domains. Through empirical investigations, theoretical analysis, and practical demonstrations, this dissertation aims to provide valuable insights and evidence-based solutions to empower organizations in successfully navigating the multifaceted challenges of modern product development.

By disseminating the findings of this research within the scientific community, it is anticipated that industry practitioners, researchers, and policymakers will benefit from the gained knowledge and practical implications, leading to enhanced product development practices and innovation in a wide range of domains.

1.3. Thesis Structure

This dissertation is structured to systematically address the research objectives and provide a comprehensive understanding of the modeling and simulation of Internet of Things (IoT) infrastructures for Cyber-Physical Energy Systems (CPES). The following subchapters outline the structure and content of this thesis:

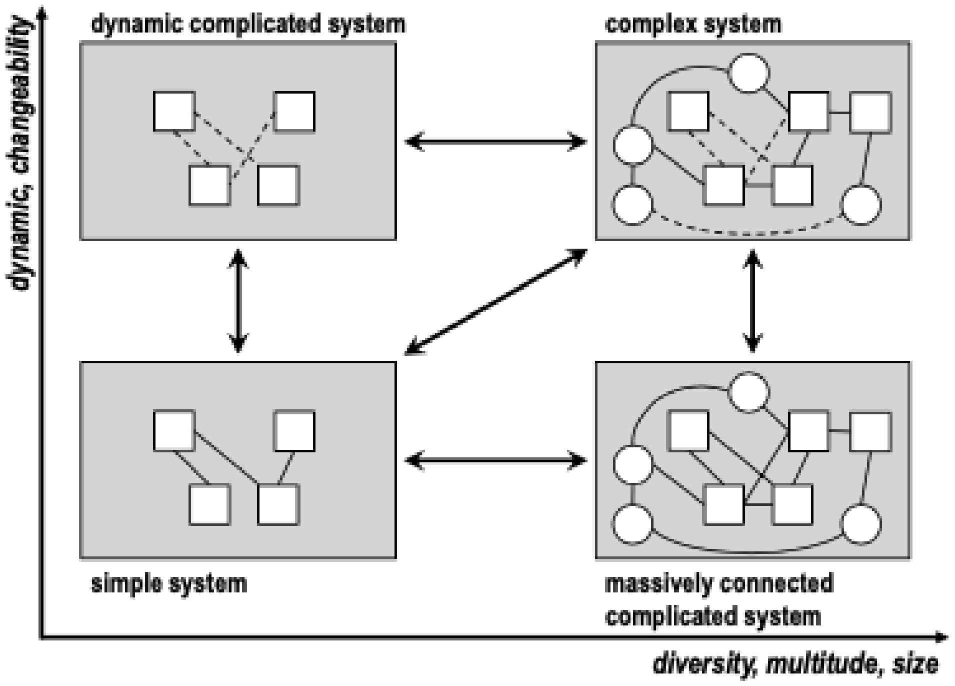

dynamic, changeability
dynamic complicated system
complex system
simple system
massively connected complicated system
diversity, multitude, size

Figure 1.4.: *System types[5]*

Introduction This subchapter provides an overview of the starting situation and sets the context for the research. It highlights the motivation behind the study and presents the thesis's target and progress over the state of the art. The subchapter concludes by outlining the overall structure of the dissertation.

Background and Fundamentals This chapter delves into the fundamental concepts and background knowledge required to comprehend the research. It covers essential topics such as the development process, requirements engineering, model-based systems engineering, and digitalization approaches. Furthermore, it explores the realm of Cyber-Physical Systems and the Internet of Things, including IoT reference architecture. The chapter also includes an examination of IoT platforms, focusing on their role in sensing the real world, IoT gateway platforms, IoT middleware, and IoT cloud platforms. Lastly, it provides an overview of the VICINITY project and introduces the concept of homomorphic encryption. State of the Art: This chapter presents an in-depth analysis of the existing literature and research pertaining to the simulation of the Internet of Things and Cyber-Physical Systems. It includes a specification of discrete event systems and provides an overview of various IoT simulators. A comparison of these simulators is also presented.

Simulation of IoT Networks This chapter focuses on the simulation of IoT networks. It introduces the concept and requirements of IoT network simulation, followed by the approach employed in this research. The chapter discusses the level hierarchy and structure, model reuse, and synchronization techniques utilized. Additionally, it explores hardware in the loop simulation, software in the loop simulation with a focus on homomorphic encryption, and human in the loop simulation.

Case Study: Smart Energy Use Case This chapter presents a detailed case study that demonstrates the application of the developed methodology and simulator. It describes the scenario model, the scenario itself, and the corresponding system model. The chapter then delves into the implementation details of the case study, including the house network, house security, house health, electronics, and house energy calculation. An exemplary system architecture is provided, showcasing the integration of homomorphic encryption micro-service. Finally, an evaluation of the case study is presented, highlighting the experimental setup and results.

Conclusions and Outlook This concluding chapter summarizes the key findings and contributions of the research. It offers a comprehensive conclusion and provides a summary of the thesis. The chapter concludes with an outlook on potential future work and research directions.

Bibliography The bibliography section lists all the references cited throughout the dissertation, enabling readers to explore the relevant sources for further study.

By following this structured approach, this dissertation aims to provide a thorough examination of the modeling and simulation of IoT infrastructures for CPES. It systematically builds upon foundational concepts, explores the state of the art, introduces novel methodologies and approaches, and showcases the practical application through a detailed case study. The subsequent chapters offer a comprehensive journey through the research process, ultimately leading to valuable insights and contributions in the field.

1.4. Thesis Target and Progress over State of the Art

The aim of this dissertation is to advance the modeling and simulation of Internet of Things (IoT) infrastructures in the context of Cyber-Physical Energy Systems (CPES). The overall objective is to enable the model-based development, verification, and validation of IoT infrastructures, facilitating earlier validation of system integration. This subchapter outlines the specific targets set for this research and highlights the progress made over the state of the art in the field.

1.4.1. Target Setting

Model-Based Development for IoT Systems The primary target is to develop a model-based approach for the early conceptual design phase of IoT systems within CPES. The approach must be capable of handling the high complexity of IoT systems, incorporating intricate interactions between cyber and physical components. The goal is to provide engineers with a methodology that supports the effective modeling of IoT systems while considering the power and energy usage of future intelligent Cyber-Physical Energy Systems.

Question 1:	How can processes, methods, and tools of IoT development be integrated into CPES Development Processes in order to improve energy consumer models?
Question 2:	How can IoT systems be described model-based in the conceptual design phase in order to fully understand their complexity?

Simulation and Emulation Framework The research aims to define a simulation and emulation framework specifically tailored to IoT infrastructures in CPES. The framework must address the unique requirements associated with power and energy usage in future intelligent systems. It should enable rapid prototyping of simulatable models during the early stages of system analysis and conceptual design, providing support for engineers working in the IoT domain.

Question 3:	How can IoT infrastructures be efficiently simulated in order to address their power and energy usage?

Integration with Real-World Scenarios The dissertation targets the integration and application of the developed methodology and simulator in realistic scenarios, particularly within the VICINITY project. The goal is to demonstrate the effectiveness of the proposed approach in developing specialized IoT infrastructures required by the project. This includes scenarios involving energy and power aspects, contributing to the dimensioning of energy networks in future smart neighborhoods.

Question 4:	How can the concepts resulting from this research be integrated and applied in realistic IoT scenarios arising from the research project VICINITY?

1.4.2. Progress over State of the Art

Model-Based Development Approach The research proposes a model-based development approach for IoT systems within CPES. This approach addresses the high complexity of IoT systems and offers a methodology for the early conceptual design phase. It leverages SysML models to accurately represent system structure and behavior, providing engineers with a mechanism to handle the intricacies of IoT systems during development. This progress represents a substantial advancement over existing approaches, which often struggle to adequately capture the complexities of IoT systems.

Simulation and Emulation Framework The research has defined a simulation and emulation framework tailored to IoT infrastructures in CPES. This framework incorporates the specific requirements related to power and energy usage in future intelligent systems. It enables engineers to rapidly prototype simulatable models during the early stages of system analysis and conceptual design. This progress represents a significant step forward in the development of simulation tools and techniques that address the unique challenges posed by IoT infrastructures.

Integration with Real-World Scenarios This dissertation demonstrates the integration and application of the developed methodology and simulator within the context of the VICINITY project. By applying the approach to realistic scenarios, particularly those involving energy and power aspects, the research has showcased the practical relevance and effectiveness of the proposed methodology. This progress contributes to the development of specialized IoT infrastructures required by the project, fostering advancements in energy management within future smart neighborhoods.

In summary, this dissertation sets specific targets for advancing the modeling and simulation of IoT infrastructures in CPES. Notably, progress has been made in developing a model-based approach for the early conceptual design phase, defining a simulation and emulation framework tailored to IoT systems, and integrating the developed methodology and simulator with real-world scenarios within the VICINITY project. These achievements represent advancements over the state of the art in the field, providing engineers with enhanced tools and techniques for the development of IoT infrastructures while considering the power and energy requirements of future intelligent Cyber-Physical Energy Systems.

1.4.3. Foundational Publications: Building Blocks of this Dissertation

As part of the groundwork for this dissertation, I have had the privilege of contributing to various publications that have shaped the trajectory of my research journey. These publications serve as foundational pillars upon which this dissertation is built, reflecting the evolution and culmination of my academic pursuits during my doctoral studies. In this subchapter, I present a comprehensive list of these publications, each representing a significant milestone in my exploration of Modeling and Simulation of Internet of Things Infrastructures for Cyber-Physical Energy Systems:

- Dalecke Š, Rafique KA, Ratzke A, Grimm C, Koch J (2022) SysMD: Towards "Inclusive" Systems Engineering. In: 2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS). IEEE, pp 1–6

- Grimm C, Wawrzik F, Jung AL-F, Luebeck K, Post S, Koch J, Bringmann O (2021) APPEL-AGILA ProPERty and Dependency Description

Language. In: MBMV 2021; 24th Workshop. VDE, pp 1–11

- Hellenthal B, Grimm C, Koch J, Breckel A, Tichy M (2021) An Eco-System for the Development of Automotive Innovation Roadmaps. VDI-Berichte 2384:285–294

- Koch J, Grimm C (2022) Cyber-Physikalische Systeme. In: Corsten H, Roth S (eds) Handbuch Digitalisierung. Vahlen, p 315

- Koch J, Wansch A, Grimm C (2022) Knowledge modeling of power grids with SysMD. In: 2022 10th Workshop on Modelling and Simulation of Cyber-Physical Energy Systems (MSCPES). IEEE, pp 1–6

- Kölsch J, Guan Y, Grimm C (2020a) Methods and Tools for Validation and Testing. In: IoT Platforms, Use Cases, Privacy, and Business Models. Springer, Cham, pp 81–98

- Kölsch J, Heinz C, Ratzke A, Grimm C (2019a) Simulation-Based Performance Validation of Homomorphic Encryption Algorithms in the Internet of Things. Future Internet 11:218

- Kölsch J, Heinz C, Schumb S, Grimm C (2018) Hardware-in-the-loop simulation for Internet of Things scenarios. In: 2018 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES). IEEE, pp 1–6

- Kölsch J, Post S, Zivkovic C, Ratzke A, Grimm C (2020b) Model-based development of smart home scenarios for IoT simulation. In: 2020 8th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems. IEEE, pp 1–6

- Kölsch J, Ratzke A, Grimm C (2019b) Co-Simulating the Internet of Things in a Smart Grid use case scenario. In: 2019 7th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES). IEEE, pp 1–6

- Kölsch J, Ratzke A, Grimm C, Heinz C, Nandagopal G (2019c) Simulation based validation of a Smart Energy Use Case with Homomorphic Encryption. In: 2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS). IEEE, pp 255–262

- Kölsch J, Zivkovic C, Guan Y, Grimm C (2020c) An Introduction to the Internet of Things. In: IoT Platforms, Use Cases, Privacy, and Business Models. Springer, Cham, pp 1–19

- Mynzhasova A, Radojicic C, Heinz C, Kölsch J, Grimm C, Rico J, Dickerson K, Garcia-Castro R, Oravec V (2017) Drivers, standards and platforms for the IoT: Towards a digital VICINITY. In: 2017 Intelligent Systems Conference (IntelliSys). IEEE, pp 170–176

- Prautsch B, Dornelas H, Wittmann R, Henkel F, Schenkel F, Kölsch J, Grimm C, Strube G (2020) AnastASICA–Towards Structured and Automated Analog/Mixed-Signal IC Design for Automotive Electronics. In: ANALOG 2020; 17th ITG/GMM-Symposium. VDE, pp 1–6
- Uwiringiyimana MM, Nandagopal G, Guan Y, Vinkovič S, Kölsch J, Heinz C (2020) IoT Platforms. In: IoT Platforms, Use Cases, Privacy, and Business Models. Springer, Cham, pp 21–49
- Zivkovic C, Grimm C, Kölsch J, Short D, Ferstl M, Denger D, Krems D, Barisic A (2020) Bringing Uncertainties into System Simulation: A SystemC AMS Case Study. In: 2020 Forum for Specification and Design Languages (FDL). IEEE, pp 1–6

Chapter 2

Background and Fundamentals

Contents

In the realm of IoT and Cyber-Physical Systems, where the intricacies of technology and connectivity continue to evolve, a solid foundation in fundamentals is imperative. This chapter delves into the essential building blocks that underpin the development process, beginning with a comprehensive exploration of requirements engineering. It then ventures into the realm of model-based systems engineering, with a focus on SysML as a critical modeling language. The chapter proceeds to dissect digitalization approaches, encompassing the ever-expanding domains of Cyber-Physical Systems and the Internet of Things (IoT). Within this context, the IoT reference architecture serves as a guiding beacon for structuring complexity in IoT ecosystems. As the IoT landscape unfolds, the pivotal role of IoT platforms emerges, orchestrating this intricate web of interconnected devices. The chapter scrutinizes the components of IoT platforms, from sensors to cloud infrastructure, and culminates in an exploration of the VICINITY project, which showcases the practical implications of IoT innovation. Additionally, the chapter delves into the realm of homomorphic encryption, a vital element in ensuring data security and privacy in IoT systems. These foundational elements collectively pave the way for a comprehensive understanding of the IoT landscape and its implications for the future of technology.

2.1. Development Process

In the context of designing and implementing complex systems, a well-defined development process is crucial to ensure the successful realization of desired outcomes. This subchapter explores the fundamentals of the development process, providing a foundation for understanding the subsequent research on modeling and simulation of Internet of Things (IoT) infrastructures for Cyber-Physical Energy Systems (CPES). The development process encompasses a series of systematic steps, from initial requirements gathering to final deployment and maintenance of a system. It provides a structured framework for guiding engineers and stakeholders through the various stages of system development, ensuring that all essential aspects are considered and addressed. In the domain of IoT and CPES, the development process plays a vital role in enabling the creation of efficient, reliable, and secure systems. It allows for the integration of cyber and physical components, leading to the distribution of energy in a coordinated manner. Additionally, a well-defined development process ensures that energy generation, transmission, distribution, and consumption are seamlessly integrated using digital communication and control technologies. The development process for IoT systems within CPES typically encompasses several key phases. These phases often include requirements engineering, design, implementation, testing, and deployment. Each phase requires careful consideration of various factors, including system functionality, performance, reliability, security, and scalability. Eliciting, assessing, documenting, and validating the system's needs are all part of the crucial initial phase known as requirements engineering. In this phase, operational, functional, and non-functional requirements for the IoT system within the CPES framework must be identified together with stakeholder needs and expectations. It lays the groundwork for later design and implementation processes. The design phase focuses on translating the requirements into a system architecture that outlines the structure, components, and interfaces of the IoT system. It involves the selection of suitable hardware and software components, as well as the specification of communication protocols, data models, and interfaces. The design phase is essential for ensuring that the system is capable of meeting the specified requirements and can be effectively implemented. The implementation phase involves the actual development and coding of the IoT system. It includes tasks such as software development, hardware integration, and configuration of communication protocols and interfaces. This phase requires adherence to coding standards, best practices, and quality assurance measures to ensure the robustness and reliability of the system. Testing is a critical phase in the development process, aimed at verifying and validating the functionality, performance, and reliability of the IoT system. It involves various testing techniques, such as unit testing, integration testing, system testing, and acceptance testing. Rigorous testing helps identify and rectify any defects or issues before the system is deployed. Finally, the deployment phase involves the installation, configuration, and commissioning of the IoT system within the CPES environment. It includes activities such as system integration, data migration, user training, and system documentation. Deployment requires careful planning and coordination to ensure a smooth transition from the development environment to the operational setting. Throughout the development process, proper project management techniques, such as scheduling, resource allocation, risk management, and stakeholder communication, are essential for successful project execution. The IoT system will be delivered on schedule, within budget, and in compliance with all requirements if there is effective project management. By understanding the fundamentals of the development process, engineers and stakeholders involved in IoT systems within CPES can navigate the complexities of system development and ensure the successful realization of their objectives. This research builds upon this fundamental understanding, aiming to enhance the development process through the application of model-based systems engineering and simulation techniques tailored for IoT infrastructures in CPES.

In the subsequent subchapters, we will explore further fundamentals and concepts that underpin the development of IoT infrastructures, including requirements engineering, model-based systems engineering, and digitalization approaches such as Cyber-Physical Systems and the Internet of Things. This comprehensive exploration will provide the necessary background knowledge to delve into the specific challenges and solutions proposed in this research for the modeling and simulation of IoT infrastructures for CPES.

2.2. Requirements engineering

Requirements engineering is a critical process within system development that focuses on eliciting, analyzing, documenting, and validating the requirements of a system. This subchapter delves into the key concepts and techniques of requirements engineering, laying the foundation for understanding its significance in the context of modeling and simulation of Internet of Things (IoT) infrastructures for Cyber-Physical Energy Systems (CPES).

The primary objective of requirements engineering is to capture and understand the needs and expectations of stakeholders and transform them into precise and unambiguous requirements. These requirements serve as the basis for system design, implementation, and testing, ensuring that the developed system meets the desired objectives and satisfies stakeholder expectations.

In the domain of IoT infrastructures for CPES, requirements engineering plays a crucial role in defining the functionality, performance, reliability, security, and other essential aspects of the system. It involves understanding the unique challenges and considerations associated with IoT systems, such as the integration of cyber and physical components, data interoperability, real-time communication, and energy efficiency.

The requirements engineering process typically involves the following key activities:

1. **Requirements Elicitation:** Engaging with stakeholders, such as end users, subject matter experts, system architects, and developers, is required for this activity to collect and establish the requirements. Techniques such as interviews, questionnaires, workshops, and observations are employed to capture the stakeholders' needs, expectations, and constraints.

2. **Requirements Analysis:** The elicited needs are examined in this exercise to make sure they are consistent, exhaustive, and feasible. The requirements are refined, prioritized, and categorized into functional and non-functional requirements. Conflicting or unclear requirements are clarified and resolved through discussion with the stakeholders.

3. **Requirements Documentation:** The needs under study are coherently laid out. In-depth descriptions of the requirements, use cases, system interfaces, and any pertinent diagrams or models are all included in the documentation. All parties concerned in the system development process can refer to the requirements documents.

4. **Requirements Validation:** This action makes certain that the specified requirements appropriately reflect the needs and expectations of the stakeholders. Validation techniques, such as reviews, walkthroughs, and prototypes, are employed to assess the quality, correctness, and completeness of the requirements. Any inconsistencies or gaps in the requirements are identified and addressed.

5. **Requirements Management:** Maintaining and monitoring requirements across the course of the system development lifecycle is known as requirements management. Changes to requirements are carefully managed, and their impact on the system design, implementation, and

testing is assessed. Proper version control and configuration management techniques are employed to ensure traceability and accountability.

Effective requirements engineering is essential for the success of IoT systems within CPES. It enables the development team to understand the system's functional and non-functional requirements, align them with stakeholder expectations, and establish a clear roadmap for system design and implementation. In-depth requirements engineering also makes it easier for stakeholders to communicate and work together, reducing risks and ensuring the success and overall quality of the system.

In the context of this research, requirements engineering serves as a crucial component in the model-based development and simulation-based verification and validation of IoT infrastructures for CPES. It ensures that the developed models accurately capture the system's requirements and align with stakeholder expectations. By integrating requirements engineering techniques within the proposed methodology, this research aims to enhance the accuracy, traceability, and efficiency of the development process for IoT systems within CPES.

2.3. Model-based Systems Engineering

Model-Based Systems Engineering (MBSE) is an approach that leverages modeling techniques and tools to support the development and management of complex systems. This subchapter delves into the key concepts and principles of MBSE, laying the foundation for understanding its significance in the context of modeling and simulation of Internet of Things (IoT) infrastructures for Cyber-Physical Energy Systems (CPES).

MBSE provides a structured and systematic approach to system development by utilizing models to capture, analyze, communicate, and validate system requirements, architecture, behavior, and other essential aspects. It allows engineers to represent complex systems in a visual and intuitive manner, promoting better understanding, collaboration, and decision-making throughout the development process.

In the context of IoT infrastructures for CPES, MBSE offers several benefits. It enables engineers to model the integration of cyber and physical components, capture system behavior and dynamics, and assess the system's overall performance and reliability. Additionally, MBSE makes it easier to analyze trade-offs, recognize dependencies, and spot potential problems or conflicts early in the development process.

The key concepts and principles of MBSE include:

1. **Models:** Models are the central artifacts in MBSE. They represent various aspects of the system, including its requirements, architecture, behavior, interfaces, and relationships. Models provide a visual representation of the system, allowing stakeholders to better understand and communicate system concepts and characteristics.

2. **Modeling Languages:** Modeling languages, such as Unified Modeling Language (UML) or Systems Modeling Language (SysML), provide a standardized notation for representing system models. These languages offer a rich set of graphical symbols and constructs that enable engineers to express system concepts, relationships, and behaviors.

3. **Model-Based Development Process:** MBSE promotes a model-centric approach to system development, emphasizing the use of models throughout the entire development lifecycle. This includes activities such as requirements modeling, architectural design, behavior modeling, verification and validation, and system integration. The use of models ensures consistency, traceability, and coherence across different development phases.

4. **Model Integration:** MBSE facilitates the integration of various models to create a holistic view of the system. This involves linking and relating different models, such as requirements models, architectural models, and behavior models, to establish traceability and coherence. Model integration enables engineers to assess the impact of changes, analyze trade-offs, and ensure the overall system consistency.

5. **Model-Based Analysis:** MBSE enables the analysis of system models to assess system behavior, performance, and reliability. Through simulations, analyses, and validations, engineers can evaluate different design alternatives, detect potential issues or conflicts, and optimize system performance. Model-based analysis helps in making informed decisions and improving the system's overall quality.

By adopting MBSE principles and techniques, engineers can enhance the development process for IoT systems within CPES. The use of models enables better understanding, communication, and collaboration among stakeholders. It facilitates the exploration of design alternatives, the identification of potential risks, and the early detection of issues, leading to improved system development outcomes.

In the context of this research, MBSE serves as a fundamental pillar in the proposed approach for modeling and simulation of IoT infrastructures for CPES. It provides the framework for capturing and representing the system's requirements, architecture, behavior, and other critical aspects. By integrating MBSE techniques with simulation-based verification and validation, this research aims to enhance the accuracy, efficiency, and effectiveness of the development process for IoT systems within CPES.

2.4. SysML - The Systems Modeling Language

SysML (Systems Modeling Language) emerges as a prominent framework within the domain of system engineering, offering a comprehensive approach to model intricate and multifaceted systems. It extends the foundational constructs of the Unified Modeling Language (UML) to cater specifically to the exigencies of system engineers. SysML provides a standardized and graphical modality for representing systems, thus facilitating the encapsulation, analysis, and effective communication of intricate system designs.

2.4.1. Blocks

At the heart of SysML are blocks, which are fundamental to modeling systems. Blocks are akin to containers that encapsulate the essential elements or components of a system. They are used to represent physical entities (like hardware components) or abstract entities (like software modules) within the system. Blocks can be used to model subsystems, systems, or even the entire system architecture.

Attributes and Properties Blocks can have attributes and properties that define their characteristics. These attributes can be used to capture information about the block, such as its size, weight, or capacity. Properties can be used to specify the values of these attributes.

Behaviors Blocks can also have behaviors associated with them. These behaviors describe how the block responds to various inputs and stimuli. For example, a block representing a sensor might have behaviors that describe how it senses and reports data.

2.4.2. Ports and Interfaces

Ports and interfaces play a pivotal role in defining how blocks interact within a system. They provide a structured way to specify how data and control signals flow between blocks.

Ports Ports are used to model the points of connection between blocks. They define the interfaces through which blocks can communicate with each other. Ports can be classified into different types, such as input ports (for receiving data), output ports (for sending data), and in-out ports (for bidirectional communication). Ports are crucial for modeling the connectivity of a system.

Interfaces Interfaces define the contractual obligations for blocks that use a port. An interface specifies the set of operations and signals that a block must support when connected to a port. Interfaces help ensure that blocks can work together seamlessly when integrated into a system.

2.4.3. Activities

SysML includes constructs for modeling the dynamic behavior of a system. Activities are used to represent the flow of actions and control processes within a system. They are instrumental in capturing how a system responds to various stimuli and events.

Actions Actions represent individual steps or operations within an activity. They can include tasks like calculations, decisions, or data processing. Actions define the discrete units of work within an activity.

Control Flow Control flow elements, such as decision nodes and control edges, are used to specify the order in which actions are executed. They define the flow of control within an activity, ensuring that actions are executed in a logical sequence.

Object Flow Object flow represents the flow of data or objects between actions. It models how data is produced by one action and consumed by another. Object flow is essential for understanding how information moves through a system.

2.4.4. Requirements

Requirements management is a critical aspect of system engineering. SysML provides constructs for capturing and managing system requirements. Requirements are statements that specify what a system must do or achieve to meet its intended purpose.

Requirement Elements SysML includes dedicated elements for representing requirements, including Requirement and Requirement Block. These elements allow you to define requirements, assign unique identifiers, and capture details such as descriptions, verification methods, and traceability links.

Traceability SysML emphasizes the importance of traceability, ensuring that requirements are linked to system components and design decisions. Traceability helps in verifying that system elements align with specified requirements throughout the development process.

2.4.5. Diagram Types

Diagrams in SysML serve as powerful tools for visualizing and communicating various aspects of a system model. They provide graphical representations that offer different perspectives on the system's architecture, behavior, and relationships. SysML supports several types of diagrams, each tailored to convey specific information about the system. Here, we explore the key SysML diagrams:

Block Definition Diagram (BDD) BDDs are fundamental for modeling the structural aspects of a system. They help in defining the composition of the system by illustrating the blocks that make up the system and how they are interconnected.

Components

- Blocks: These are central to BDDs. Each block represents a system component, whether it's a physical entity like a sensor or an abstract concept like a control algorithm.
- Ports: Ports are depicted on blocks to show how they interact with other blocks. Ports represent the interfaces through which data and control signals flow.
- Connectors: Connectors establish relationships between blocks. They represent the connections or associations between blocks, indicating how they collaborate.

Use Cases

- System Architecture: BDDs are like architectural blueprints, providing a high-level view of the system's structure. They are useful for stakeholders to understand how various components fit together.
- Interface Design: BDDs help in specifying interfaces (ports) and their connections, ensuring that blocks can communicate effectively.
- Hierarchy: BDDs can illustrate the hierarchical structure of a system, showing how blocks are organized into subsystems.

Internal Block Diagram (IBD) IBDs offer an inside look at the internal structure of a block. They detail the components and their relationships within a block, allowing for a deeper understanding of its composition.

Components

- Parts: Parts represent the internal components of a block. They could be other blocks, connectors, or value types.
- Connectors: Connectors in IBDs illustrate the relationships and interactions between parts within a block.
- Ports: If a block in an IBD has ports, they can be shown along with their connections to external blocks.

Use Cases

- Component Detail: IBDs provide a detailed view of a block's internal composition. This is valuable when examining how a complex block is constructed from smaller parts.
- Subsystem Exploration: IBDs are useful for diving into the structure of subsystems defined within blocks, helping engineers understand how these subsystems interact.
- Signal Flow: IBDs can illustrate how data and signals flow between parts within a block, aiding in the analysis of internal data exchange.

Activity Diagram Activity diagrams are used to model the dynamic behavior of a system, showcasing how actions and processes are performed over time.

Components

- Actions: Actions represent individual steps or tasks within the activity. They could be anything from calculations to decision-making processes.
- Control Flow: Control flow elements like decision nodes and control edges define the order in which actions are executed, creating a flowchart-like representation of the process.
- Object Flow: Object flow depicts how data or objects move between actions, demonstrating how information is processed.

Use Cases

- Process Modeling: Activity diagrams are excellent for modeling and understanding complex processes and workflows within a system.
- Behavior Analysis: They aid in analyzing how actions interact and influence each other during system operation.
- User Interaction: Activity diagrams can also represent user interactions with the system, making them valuable for designing user interfaces.

Sequence Diagram Sequence diagrams focus on the temporal aspects of system behavior, illustrating how objects (blocks) interact over time through message exchanges.

Components

- Lifelines: Lifelines represent objects involved in the interaction. Each lifeline corresponds to a block or entity, showing its existence over a period.
- Messages: Messages are used to depict interactions between lifelines. They can represent method calls, data exchanges, or control signals.
- Activation Bars: Activation bars indicate the time during which an object is active and processing messages.

Use Cases

- Interaction Modeling: Sequence diagrams are ideal for modeling the sequence of interactions between system components, aiding in understanding message exchanges.
- Timing Analysis: They help in analyzing the timing and ordering of events during system operation.
- System Integration: Sequence diagrams can show how different blocks collaborate in complex systems, highlighting communication patterns.

Requirement Diagram Requirement diagrams provide a structured view of system requirements and their relationships. They help manage and trace requirements throughout the development process.

Components

- Requirements: Requirements are represented as individual elements in the diagram. Each requirement may have attributes like a unique identifier, a description, and verification methods.
- Relationships: Relationships between requirements are depicted to show how they depend on or relate to each other. Common relationships include *satisfy*, *derive*, and *verify*.

Use Cases

- Requirement Management: Requirement diagrams serve as a central repository for system requirements, ensuring that they are well documented and traceable.
- Impact Analysis: They help in understanding how changes in requirements can affect other parts of the system.
- Verification and Validation: Requirement diagrams assist in planning and executing verification and validation activities to ensure that the system meets its requirements.

Understanding and effectively utilizing these SysML diagrams is crucial for system engineers to communicate system designs, behaviors, and requirements clearly and comprehensively. Each diagram type offers a unique perspective on the system, facilitating the modeling and analysis of complex systems in a structured manner.

2.5. Digitalization Approaches

This chapter explores digitalization approaches that underpin the development of Internet of Things (IoT) infrastructures for Cyber-Physical Energy Systems (CPES). It examines key concepts such as Cyber-Physical Systems, the Internet of Things, and the IoT reference architecture. Through this exploration, we gain a comprehensive understanding of the digitalization landscape, providing valuable insights for the modeling and simulation of IoT infrastructures within the CPES context.

2.5.1. Cyber-Physical Systems

Cyber-Physical Systems (CPS) represent a paradigm that integrates physical entities with computational and communication systems. CPS enable seamless interaction and collaboration between the physical and digital realms, bridging the gap between them. In the context of Internet of Things (IoT) infrastructures for Cyber-Physical Energy Systems (CPES), CPS play a crucial role.

At the core of CPS is the tight integration of physical processes, computational elements, and communication networks. They combine real-time sensing, actuation, and computation to enable intelligent decision-making and advanced functionalities. CPS facilitate the integration of energy generation, transmission, distribution, and consumption processes with computational and communication systems within CPES.

CPS enable real-time monitoring, control, and optimization of energy flows, leading to enhanced energy efficiency, reliability, and sustainability. They leverage computational elements such as sensors, actuators, embedded systems, and control algorithms to acquire, process, analyze, and actuate data from physical processes. Communication networks enable the exchange of information between the physical and digital components, enabling coordination and collaboration.

The integration and interaction between the physical and digital components of CPS create a dynamic feedback loop. The physical processes influence the computational elements, while the computational elements affect the physical processes through actuation and control. This integration allows for real-time monitoring, analysis, and control of CPES.

By embracing CPS principles, engineers can design and develop IoT infrastructures that effectively integrate cyber and physical components within CPES. CPS form the foundation for advanced functionalities such as real-time monitoring, predictive maintenance, energy optimization, and intelligent decision-making. The seamless interaction between the cyber and physical realms fosters enhanced energy management, reliability, and sustainability in CPES.

2.5.2. Internet of Things

This chapter provides a comprehensive introduction and overview of the Internet of Things (IoT). It is essential to comprehend the evolution of the IoT concept, which has transformed from the World Wide Web to a network primarily connecting machines and objects, rather than humans. This transformation reflects the changing landscape of connectivity and highlights the expanding role of automated communication.

To facilitate a clear understanding of IoT, we offer an introductory discussion on the technologies and standards relevant to machine-to-machine (M2M) communication within the IoT framework. This discussion serves as a foundation for comprehending the technical underpinnings of IoT systems.

A fundamental aspect of the chapter explores the architecture of IoT systems. This architecture delineates the key components of IoT and elucidates how they interact to form a cohesive and functional system. Understanding this structure is pivotal to grasping the inner workings of IoT technology.

Moreover, the chapter delves into various use cases where IoT systems excel. It elucidates how IoT contributes to enhanced efficiency, automation, and decision-making capabilities across different domains. These real-world scenarios underscore the practical advantages of IoT implementation.

However, the chapter also recognizes the challenges entailed in the development and deployment of IoT systems. While IoT holds significant promise, it is not without its complexities and obstacles. By acknowledging these challenges, we prepare the groundwork for a comprehensive exploration of IoT's significance and its potential impact, particularly in the context of Cyber-Physical Energy Systems. Through this introductory chapter, readers will gain a solid foundation for the subsequent in-depth analysis of IoT in the energy sector. Please note that certain portions of the content and findings presented in this chapter have been previously disseminated in [6] and [7], and have been thoughtfully incorporated into this dissertation to provide a comprehensive and coherent perspective on the subject matter.

From the World Wide Web to the Internet of Things

The inception of the internet can be traced back to the 1980s, a pivotal period when computer networks began gaining widespread popularity. During this era, the primary function of these networks was to facilitate user access to documents stored on computers located around the world. Users typically utilized protocols like File Transfer Protocol (FTP), particularly in academic settings. In parallel, proprietary networks such as America Online (AOL) and the German Bildschirmtext (BTX) system emerged, catering to private users' communication needs. However, these proprietary networks faced challenges related to standardization, content heterogeneity, and limited accessibility.

To address these limitations, an early solution named Gopher surfaced as a popular approach. Gopher introduced innovative features like generated menus, formatted text, and document cross-references [8]. Although Gopher offered a more organized and user-friendly way to access information, it eventually gave way to a more transformative vision.

Concurrently, Tim Berners-Lee was formulating a visionary concept – a ”World Wide Web” characterized by interconnected documents. In April 1993, this vision became a reality when the World Wide Web, offering hypertext capabilities enabled through the Hypertext Markup Language (HTML), was made accessible to the public at no cost. This historic milestone marked the birth of the internet and the World Wide Web as they exist today.

From a technical standpoint, it's crucial to distinguish between the terms ”internet” and ”WWW.” The term ”internet” pertains to the underlying communication protocols that facilitate global connectivity. These protocols include technologies like Ethernet and Wireless Local Area Network (WLAN) for establishing physical connections, as well as Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6), Transmission Control Protocol (TCP), and Internet Control Message Protocol (ICMP) for efficiently transporting data across the network [9].

In contrast, ”WWW” refers to the application layer protocols that facilitate the exchange of documents in HTML format within the World Wide Web. Notable protocols in this context include the Hypertext Transfer Protocol (HTTP) and its secure counterpart, Hypertext Transfer Protocol Secure (HTTPS) [10]. These protocols are fundamental to the seamless retrieval and display of HTML documents, and they have played a pivotal role in shaping the user experience on the World Wide Web.

This historical evolution from early computer networks to the birth of the World Wide Web lays the groundwork for understanding the subsequent transformation of the internet into the Internet of Things (IoT). In the following sections, we will delve deeper into the IoT's emergence, architecture, and transformative impact on diverse sectors.

Rise of the Internet of Things With the advent of the World Wide Web, it quickly became apparent that computers were no longer exclusively tools for human interaction with information. Machines of various types also began utilizing the internet to exchange data and offer services. A notable early example of this trend can be traced back to 1982 when a Coca Cola machine, connected to a refrigerator via the internet, could report on the availability of cold drinks [11]. This seemingly mundane event foreshadowed a profound transformation.

In 1991, Mark Weiser articulated the concept of "ubiquitous computing," envisioning a future where computers seamlessly integrated into everyday devices would function imperceptibly, without being recognized as standalone computing devices [12]. This vision laid the groundwork for a paradigm shift in computing.

The term "Internet of Things" (IoT) has since gained prominence as a descriptor for the networking of smart objects via the internet. The IoT facilitates communication and service provision among these objects without direct human involvement. Coined by Kevin Ashton in 1999, initially within the context of Radio-frequency identification (RFID) tags, the IoT has been officially defined by the International Telecommunication Union (ITU) as *"a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies."* [13, 14].

The proliferation of connected devices within the IoT has grown at an astonishing rate, with projections indicating that this number will reach 41.6 billion devices by 2025, according to the International Data Corporation (IDC) [15]. As these devices within the complex IoT ecosystem employ diverse technologies, a fundamental question emerges: how can these devices effectively communicate and interact with each other? Addressing this challenge has led to the development of numerous IoT platforms available in the market, each offering its own set of solutions.

However, the process of selecting the right IoT platform introduces another critical question: how can we ensure that the chosen platform aligns with our specific needs and objectives? One valuable approach to addressing this issue is through the examination of IoT reference architectures. These architectures serve as comprehensive frameworks for assessing the capabilities of existing IoT solutions and guiding the development of new products and services in this

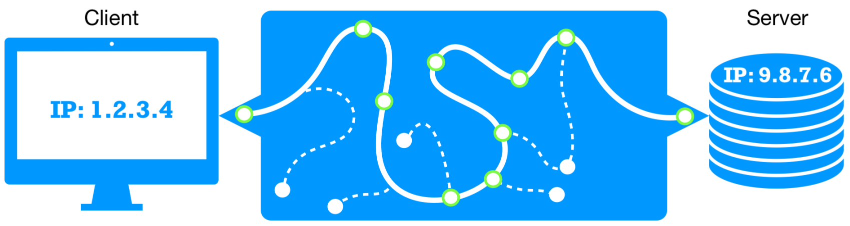


Figure 2.1.: *Example of a connection between client and server including the route*

dynamic and ever-expanding landscape. In the subsequent sections, we will delve deeper into the intricacies of IoT architectures and their role in shaping the future of interconnected devices.

Communication in the IoT: Standards and Protocols The Evolution of Internet Connectivity

The transformational power of the internet began to unfold approximately two decades ago when personal computers and internet connectivity became ubiquitous. This widespread adoption marked a profound shift in the field of computer science. It ushered in an era where information was globally accessible to people at any time with minimal latency. The internet operates on a set of foundational principles, including the critical concept of data transport through node-to-node connections, which requires intricate routing mechanisms to ensure the efficient delivery of data to its intended destination.

One remarkable characteristic of the internet is its decentralized architecture, allowing every internet-ready device to establish connections with one another. This decentralized structure is exemplified in Figure 2.1, where any node can seamlessly switch roles, acting as both a service provider (server) and a service consumer (client). This peer-to-peer communication underpins the internet's robustness. Another fundamental principle of the internet is its commitment to treating all transmitted data equally, regardless of its source or content.

To enable smooth internet usage, a suite of standard protocols has been established, including the well-known TCP/IP (Transmission Control Protocol/Internet Protocol) and UDP (User Datagram Protocol). These protocols are the backbone of reliable and efficient communication between devices connected to the internet.

However, the direct application of these internet standards to the context of the Internet of Things (IoT) and Machine-to-Machine (M2M) communications poses specific challenges. Interconnected IoT devices often operate with constrained resources, requiring specialized protocols that support low-power communication. In the forthcoming sections, we will provide a concise overview of some of the widely adopted protocols used in M2M/IoT communication. These protocols have been tailored to address the unique requirements of IoT and M2M systems, ensuring efficient and effective communication within the context of the interconnected world of devices.

M2M/IoT Communication Protocols The domains Machine-to-Machine (M2M) and Internet of Things (IoT) communication have given rise to a suite of specialized protocols, each meticulously crafted to meet the distinct demands of interconnected devices. One such protocol is MQTT (Message Queue Telemetry Transport), which operates on a publish/subscribe model. MQTT has been engineered to streamline communication among devices, especially those with power constraints, making it exceptionally well-suited for IoT devices.

Within the realm of MQTT-based communication, devices adopt the roles of clients, connecting to an MQTT broker functioning as a server. The broker's primary responsibility is to route messages efficiently between clients. MQTT leverages the publish/subscribe paradigm, enabling devices to publish messages to the broker, which subsequently delivers these messages solely to subscribed clients with an interest in the respective topic. This decoupling of publishers and subscribers based on message topics eliminates the necessity for direct awareness of each other's existence. MQTT shines in scenarios involving low-bandwidth and high-latency networks, typical characteristics of IoT networks. It frequently finds application in centralized IoT networks, where a central server governs communications between various devices. A prominent use case for MQTT is in the communication between IoT gateways and the devices connected to them.

Another notable player in the M2M/IoT communication arena is CoAP (Constrained Application Protocol). CoAP is a specialized web transfer protocol meticulously designed to cater to resource-constrained devices on the internet. It operates as a service layer protocol and is formally defined in RFC 7252. CoAP's versatility is further enhanced by the contributions of the ETF CoRE (Constrained RESTful Environments) working group, which has extended the protocol to better accommodate the nuances of M2M/IoT communications.

CoAP typically operates over UDP (User Datagram Protocol) at the transport layer, although it can optionally be bound to DTLS (Datagram Transport Layer Security) to bolster security for M2M communications. CoAP adopts a RESTful model akin to HTTP, wherein servers assign a Unique Resource Identifier (URI) to each resource. Clients can subsequently employ HTTP methods like GET, PUT, POST, and DELETE to access and interact with these resources [16]. Similar to HTTP, CoAP offers support for various data formats such as XML (Extensible Markup Language) and JSON (JavaScript Object Notation) for data representation and exchange. This adaptability makes CoAP a versatile choice for diverse M2M/IoT communication scenarios.

TCP/IP At the very core of modern internet communication lies the synergistic partnership between TCP (Transmission Control Protocol) and IP (Internet Protocol). These two protocols are instrumental in facilitating packet-oriented data transfer, forming the bedrock of the internet by addressing fundamental communication requirements. In the early 1980s, the Defense Ad-

Task	Standard
Logical Addressing	IP [17]
Fragmentation	IP [17]
Routing	IP [17]
Error Control	TCP [9]
Flow Control	TCP [9]
Application Support	TCP [9]

Table 2.1.: *Differentation of tasks of TCP and IP*

vanced Research Projects Agency (DARPA) assumed a pivotal role in crafting the standards for TCP/IP. This endeavor was primarily geared toward the development of a prototype network, known as ARPANET. DARPA's pioneering efforts culminated in the creation of RFC 760 [17], which meticulously outlines the specifications for IP, and RFC 793 [9], the defining document for TCP.

The process of internet communication unfolds as data embarks on a journey, traversing myriad network nodes to ultimately reach its intended destination. Along this arduous path, data must undergo fragmentation into fixed-size data packets to ensure successful transmission (see Fig. 2.1). To guarantee reliable message exchange among interconnected systems, mechanisms for detecting packet loss, duplicates, and reordering are indispensable. These critical facets of internet communication are exhaustively addressed and standardized within the ambit of the RFC standards, as summarized in Table 2.1.

A TCP/IP connection epitomizes a lossless communication channel, enabling the seamless transmission of data from a client endpoint to a server endpoint, even as it traverses an intricate network of nodes. This attribute renders TCP/IP particularly well-suited for centralized networks where a server plays a pivotal role in orchestrating communications between clients. The endpoints themselves, referred to as sockets, are provided by the respective operating systems and are accessible through a socket application programming interface (API) (for a more in-depth discussion, see Section 2.5.2).

One of the hallmark features of TCP is its unwavering commitment to ensuring the reliable stream of data. It achieves this feat through the employment of a unique three-way handshake mechanism, an essential component in the establishment and maintenance of a connection between the parties engaged in communication [18]. This dedication to data reliability has propelled TCP into a linchpin of internet communication, underpinning countless digital interactions worldwide.

UDP In the intricate tapestry of internet communication, the User Datagram Protocol (UDP), introduced in 1980 through RFC 768, assumes a distinctive role. Situated at the Transport layer, UDP builds upon the bedrock of the underlying IP protocol. It offers a conduit for transmitting messages, known as datagrams, delivering a best-effort service characterized by a unique

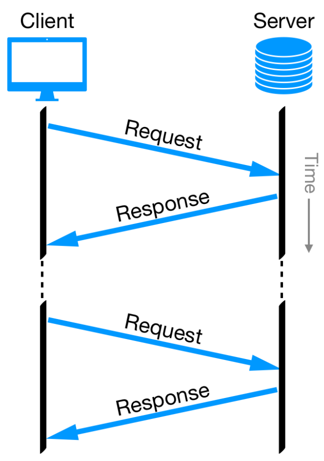


Figure 2.2.: *Sequence diagram of HTTP requests and responses over the time*

set of features. Unlike its counterpart, TCP, UDP functions as an unreliable datagram protocol. It forgoes mechanisms for delivery control, duplicate protection, or ordering functionality. In essence, UDP operates on the premise of delivering data expediently, without the overhead of the elaborate coordination and handshakes required by TCP. Consequently, it emerges as the preferred choice for applications prioritizing speed over meticulous reliability.

A UDP datagram, the fundamental unit of data in UDP communication, is encapsulated within a single IP packet. This design imposes a constraint on the maximum payload size. To initiate the transmission of a UDP datagram, various fields within the UDP header, including the crucial Port Control Information (PCI), are meticulously configured. Subsequently, the data, accompanied by the header, embarks on its journey through the IP network layer, poised for swift transmission [19].

UDP's distinctive design caters to scenarios where rapid data transfer outweighs the need for exhaustive reliability measures. It has found its niche in applications demanding low-latency connections, where the swiftness of communication takes precedence over meticulous error-checking and data coordination.

HTTP: A Stateless Communication Protocol The Hypertext Transfer Protocol (HTTP) stands as a cornerstone of the web, facilitating the seamless exchange of information across the digital landscape [18]. The term "hypertext" signifies text containing embedded links, commonly referred to as hyperlinks, which enable users to traverse the interconnected web of information. HTTP functions within a message-based model, orchestrated by both clients and servers. Clients initiate this communication by dispatching requests to servers, which respond with corresponding answers, forming a dialogue reminiscent of human interaction.

Each interaction, be it a request or response, comprises two primary components: a header and a payload. Headers are exclusively reserved for textual information, whereas payloads can encompass both text and binary data. In

Status number	Message
1xx	Informational response
100	Continue
2xx	Success
200	OK
201	Created
202	Accepted
3xx	Redirection
301	Moved Permanently
4xx	Client Errors
403	Forbidden
404	Not Found

Table 2.2.: *An extract of HTTP status codes of RFC 2616 [20].*

Figure 2.2, the client initiates this exchange by forwarding a request header replete with parameters, codec details, and the desired resource path. Notably, clients have the liberty to incorporate a payload directly within the request. In return, the server reciprocates by delivering data - often an HTML document - as the payload, accompanied by a precise response status (as delineated in Table 2.2). It is imperative to underscore that HTTP operates in a stateless manner, necessitating clients to furnish all requisite information with each request due to the absence of persistent session context.

HTTP, as specified in the widely adopted version 1.1, endows request headers with a repertoire of methods, serving as unequivocal directives. Among these, the most frequently employed methods include GET, PUT, DELETE, and POST, each instrumental in shaping the nature of interactions between clients and servers, thus ensuring effective communication.

- The GET Method: This method orchestrates requests aimed at retrieving specific data corresponding to the request-URI. Its primary function revolves around information retrieval, allowing clients to access and peruse resources without inducing alterations.

- The PUT Method: Triggering this method compels the server to modify a designated resource aligned with the request-URI. The alterations are based on the data encapsulated within the request body of the HTTP PUT request, empowering clients to update or overwrite existing resources.

- The DELETE Method: Deploying the DELETE method prompts the server to expunge the resource identified by the request-URI. Clients wielding this method can initiate the removal of specific resources, effecting the deletion of data housed on the server.

- The POST Method: In contrast, the POST method serves as a conduit for creating a new resource on the server. By transmitting a request

Component	**Significance**
host	The IP address or a registered hostname of a host located on a server, serving as the fundamental point of access within a Uniform Resource Identifier (URI).
port	The numerical port number on the host, representing the designated endpoint for communication.
path	A sequence of segments, hierarchically organized and separated by slashes, denoting the specific location of the resource within the host.
query	An optional component, typically initiated with a "?" character, providing supplementary data or parameters that can be employed for various purposes, such as search or data retrieval.
fragment	An optional component, often introduced by a "#" character, serving for local reference and processing within a web browser; it enables additional identification or handling of specific sections within the resource hosted on the server.

Table 2.3.: *Decomposition of an URI*

body within the HTTP POST request, clients convey their intention to generate a resource associated with the request-URI, thereby instigating the creation of new data on the server.

These HTTP methods, intrinsic to the protocol, underpin a spectrum of interactions between clients and servers, facilitating agile data manipulation and resource administration across the World Wide Web.

Decoding Universal Resource Identifiers (URIs) In the digital realm, gaining access to an abstract or physical resource necessitates the use of a Universal Resource Identifier (URI). This critical element comprises a string of characters partitioned into distinct components using predefined operators. Although the generic URI syntax exhibits a hierarchical structure, each constituent element serves a distinct purpose in facilitating resource retrieval.

A typical URI syntax encompasses the following elements:

```
URI = scheme "://" host : port "/" path "?" query "#" fragment
```

The meaning of each component is elucidated in Table 2.3.

Example: http://10.0.0.1:80/news/?q=test#X_new

Here, the blue-marked components pertain to communication, as elucidated in the TCP/IP section. Conversely, the brown-marked components are integral to the HTTP request.

RESTful API: Navigating M2M Communication API, an acronym for Application Programming Interface, plays a pivotal role in M2M (Machine-to-Machine) communication by offering a standardized framework for client-server interactions. In the realm of distributed connected systems, communication while sharing compatible data is a foundational task, resulting in a multitude of paradigms for remote procedure calls.

Notably, REST-API (Representational State Transfer - Application Programming Interface) stands out as an implementation approach characterized by a RESTful architecture, rather than a specific protocol or standard. REST, which stands for Representational State Transfer, leverages standardized URIs, HTTP, and typically employs JSON as a data format. This approach adheres to six fundamental principles.

Unlike alternative API design strategies, REST does not encode particular methods within the URI. Instead, it navigates to the resource via the URI and executes the provided HTTP request therein. In essence, any URI featuring static content is inherently compatible with the REST architecture due to the intrinsic similarities between REST and HTTP.

It's imperative to recognize that a RESTful architecture isn't optimized for frequent content changes within a short time span, as each update necessitates a new HTTP request [21].

JSON: A Lightweight Data Exchange Format JSON, which stands for JavaScript Object Notation, is described as a *"lightweight, text-based, language-independent data exchange format"* in RFC 4627. It serves as a compact set of formatting rules designed to facilitate the transportability of structured data. JSON's primary function is to provide a textual format for serializing structured data, and it has its roots in JavaScript, a prominent programming language.

At its core, a JSON file is constructed from two fundamental elements: objects and arrays. JSON's flexibility allows objects or arrays to be effortlessly constructed from JSON strings. Objects, in JSON terminology, represent collections of "name" and "value" pairs, and they do not possess a predefined order. Conversely, arrays consist of an ordered sequence of "value" pairs.

The constituents of a "value" pair within JSON are versatile, encompassing strings, numbers, booleans, null values, nested objects, or arrays. In contrast, a "name" pair is invariably a string [22]. JSON's simplicity, universality, and compatibility with multiple programming languages have contributed to its status as a prevalent data interchange format in various domains.

The elegance of JSON lies in its ability to represent complex data structures in a readable and concise textual format, making it a preferred choice for data exchange between diverse software systems and components. Its straightforwardness and adaptability render it invaluable in scenarios where human readability is essential, such as configuration files, APIs (Application Programming Interfaces), and data transmission between web servers and clients.

JSON's versatility and widespread adoption underscore its significance in contemporary computing, making it an essential component in the communication and interoperability of diverse systems [22].

2.5.3. IoT Reference Architecture: Structuring Complexity

As the Internet of Things (IoT) continues to evolve and grow in complexity, there is a pressing need to organize its architecture into distinct layers. This structural approach enables a more systematic understanding of the IoT ecosystem and facilitates the development and deployment of IoT solutions. Scholars such as Zhu et al. [23] and Chen et al. [24] have made significant contributions by introducing a three-layered framework for constructing an IoT reference architecture. These three layers are:

Things layer The things layer serves as the foundation of the IoT architecture, where smart objects, sensors, and devices actively participate. Their primary role is to generate data, capturing information from their surroundings. To achieve this, a variety of communication standards come into play, including RFID (Radio-Frequency Identification), ZigBee, Bluetooth, and 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks). However, these technologies often use proprietary or specialized protocols for communication. To facilitate interoperability and communication with higher layers, this data must be converted into a standard communication protocol. This essential protocol conversion is typically handled by universal devices known as ”IoT gateways.” These gateways act as intermediaries between the things layer and the network layer, ensuring that data from various sources can be efficiently transmitted upwards through the architecture.

Network layer The network layer, also referred to as the communication or connectivity layer, plays a pivotal role in transporting data collected from the things layer to remote locations. This layer employs well-established Internet communication protocols like Ethernet, Wi-Fi, and GPRS (General Packet Radio Service) to establish connections and transfer data. It serves as the backbone of the IoT infrastructure, ensuring that information collected by smart objects can be relayed to centralized systems, data centers, or other relevant destinations. By utilizing these robust and widely adopted communication standards, the network layer forms the critical bridge between the IoT's sensory foundation and the upper layers responsible for processing and application.

Application layer Sitting atop the IoT architecture is the application layer, which directly interfaces with end-users and various applications. This layer serves as the point of interaction between humans and the IoT ecosystem. Depending on the specific needs and requirements of users and organizations, the application layer provides a diverse array of services and functionalities. This layer is exceptionally versatile and spans multiple application domains, including smart buildings, smart homes, smart transportation systems, and smart energy grids. In essence, the application layer harnesses the data and capabilities of the underlying IoT infrastructure to deliver practical solutions and services that enhance efficiency, automation, and decision-making across a wide spectrum of contexts.

While these three primary layers provide the foundational structure of the IoT architecture, the ever-growing heterogeneity and complexity of IoT systems have necessitated the introduction of additional layers. As IoT networks expand and diversify, the sheer volume of data generated by devices utilizing various technologies becomes a significant challenge to manage effectively. To address these emerging needs and complexities, a fourth layer, known as the service management layer, comes into play [25].

Service management layer The service management layer in the IoT architecture plays a multifaceted role in ensuring the efficient functioning of IoT networks. It employs proper identifiers, such as URIs (Uniform Resource Identifiers), to enable service discovery and manages services and their associated requests. This layer is responsible for the storage and management of data collected from the network layer, transforming it into semantically meaningful information. Furthermore, it leverages semantic technologies to carry out information discovery, enabling intelligent decision-making processes. The service management layer extends its influence up to the application layer, ensuring that processed data and insights are seamlessly integrated into user-facing applications and services.

As IoT networks mature and expand, they open up new avenues for creating successful business models. To harness the full potential of an IoT network and transform it into a viable business opportunity, the introduction of a fifth layer, often referred to as the business layer [25], becomes imperative.

Business layer The business layer, the fifth and topmost layer in the IoT architecture, assumes a pivotal role in orchestrating the various components and functionalities of an IoT system to drive business outcomes and create value. This layer is primarily responsible for the strategic management of applications, overseeing the data generated by these applications, and extracting valuable business insights from this data.

In essence, the business layer acts as the nexus between the technical infrastructure of the IoT, the applications that run on it, and the overarching business objectives. It facilitates the alignment of IoT initiatives with broader business strategies, ensuring that the technology is leveraged to achieve tangible business goals.

By integrating the capabilities of the lower layers, including data storage, data analysis, service management, and semantic services, the business layer enables organizations to harness the full potential of their IoT investments. It not only manages the applications that utilize IoT data but also provides the frameworks and tools necessary for deriving actionable insights from this data.

This holistic five-layer IoT architecture (refer to Figure 2.3), as proposed by Fuqaha et al. [25], and further refined by Silva et al. [26], encompasses all key domains essential for the development and operation of a comprehensive IoT system. These domains include the object domain (sensors and devices), the network or connectivity domain, the middleware domain (encompassing data storage, data analysis, service management, and semantic services), the application domain, and finally, the business domain. By systematically addressing these domains, organizations can build robust and effective IoT solutions that deliver real value across various sectors and industries.

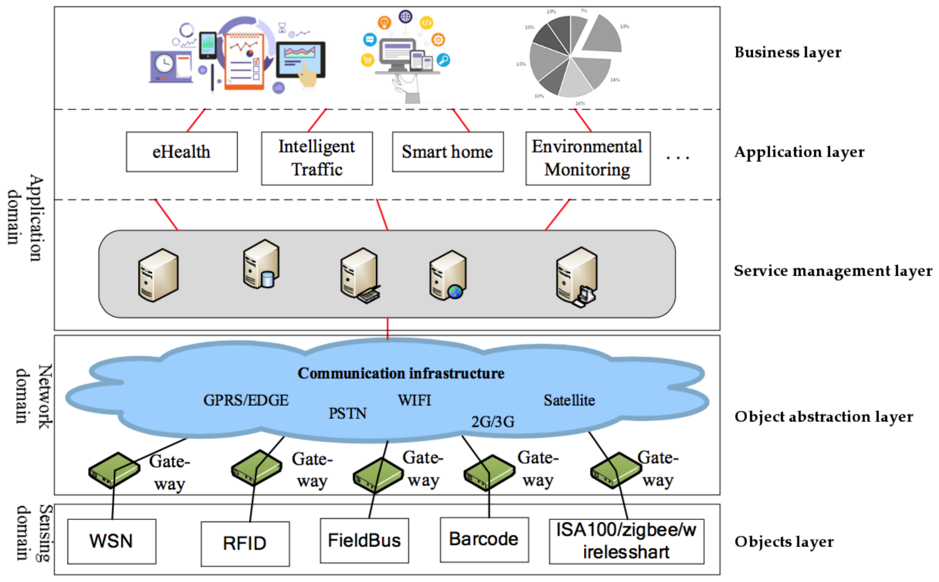


Figure 2.3.: *Five-layer IoT architecture [2]*

Use Cases of the IoT: Beyond a Fridge with Webcam

IoT Services: How much? And where? The Internet of Things (IoT) is not merely a network of interconnected gadgets; it represents a paradigm shift in how we interact with and utilize technology. To gain a deeper understanding of what makes an IoT use case valuable and where its true potential lies, let's explore two simple real-life examples:

(1a) *Anne is at work and realizes she forgot to turn off the lights at home. Fortunately, she can remotely control an IoT-enabled light bulb using a smartphone app.*

In example (1a), Anne uses the internet to control a device without her physical presence. Another example could be:

(2a) *Anne is shopping for party supplies but can't remember which drinks she needs. She checks her fridge, which displays the contents, helping her identify the missing items.*

In these examples, Anne leverages the internet to interact with devices without physical presence. However, these scenarios represent only basic forms of communication. They lack essential elements like data collection, intelligent decision-making, and autonomous control.

The IoT extends beyond remote control and data retrieval. It enables the collection, aggregation, and analysis of vast amounts of data from various sources, including sensors, personal information, and social media. This data allows machines to make informed decisions based on well-defined scenarios, moving beyond mere responsiveness to human input, as seen in (1a) and (2a).

Consider a more advanced example:

(1b) *Anne is leaving for work, and the IoT, based on data from multiple sources, determines that there are no occupants at home. Consequently, it automatically turns off the lights.*

In (1b), Anne benefits from time and resource savings. The IoT's advantages include the ability to control devices remotely, receive real-time data without human intervention, and automatically manage resource utilization by aggregating data and providing decision support.

However, the real transformative potential of IoT becomes evident when data from multiple sources are combined:

(2b) *Anne is driving her autonomous car, which decides to take her to a store because her refrigerator indicates a shortage of drinks for her upcoming party. It selects a local store and displays the beverages liked by her Facebook friends.*

These examples illustrate the power of data aggregation and decision-making in IoT services. While the benefits are substantial, they also raise important considerations, including privacy and control over personal information. As IoT continues to evolve, finding the right balance between convenience and data security becomes crucial.

Use cases of the IoT

The Internet of Things (IoT) finds applications in a wide range of domains, enhancing efficiency, automation, and decision-making processes. The following is a selection of popular domains where IoT technologies are employed:

Energy The concept of the Internet of Energy, a convergence of distributed energy generation systems and IoT, transforms the traditional electrical grid into a smart grid. This transformation encompasses every facet of the grid, from electrical devices and sensors to high-level applications and use cases driven by data. One intriguing aspect is how interconnected distributed energy systems within a neighborhood can collectively negotiate their potential energy flexibility, encompassing both distributed generation and consumption. This collective dynamic Demand Side Management (DSM) strategy opens the door to more efficient energy utilization.

Consider a municipal setting with a community-scale microgrid serving a cluster of municipal buildings, supplying their power and heating requirements. Within this microgrid-enabled energy network, precise control of energy flows is essential to minimize costs and maximize technical efficiency. This effective management hinges on well-informed decision-making processes. These decisions can be expedited by amalgamating data from energy system information models and sensor data, reducing the time required for decision elaboration and ensuring optimal energy utilization.

Buildings and Homes Efficient resource management, resource consumption, and predictive operations are paramount for both residential homes and commercial buildings. Building automation and management have witnessed a surge in current trends, with the integration of various wired and wireless communication systems, such as KNX, ModBus, and MBus. These systems are often interconnected with the building's operating system. Additionally, user-developed technologies, covering aspects like safety, health, and climate, are becoming increasingly prevalent, often operating independently from the building's infrastructure.

For smart buildings, a crucial consideration is the ability to capture data from these diverse sources comprehensively, with clear semantic understanding. This data should also be made accessible to other systems requiring it for their business processes. To illustrate this data flow, consider a building context. However, it's equally vital that the same information elements are available for processes and actors operating outside of the building environment, including those in energy, smart community development, health, and transportation sectors.

Governments and regulatory bodies worldwide are taking proactive measures to enhance the energy efficiency of both commercial buildings and residential neighborhoods. They are leveraging IoT technologies to unlock new economic opportunities and benefits for all relevant stakeholders. These technologies are expected to not only enhance the usability, capacity, and cost-effectiveness of homes and buildings but also significantly boost their energy efficiency, aligning with global sustainability goals [27].

Health The healthcare landscape is facing escalating challenges, primarily due to an aging population characterized by increasing rates of chronic illnesses like obesity, dementia, and hypertension. Effectively managing healthcare in such a scenario requires innovative solutions. E-health and assisted living technologies designed for the elderly and individuals with special needs offer a promising approach. These technologies aim to create decentralized and user-friendly platforms that bridge end-users with their families, healthcare professionals, and assistance providers.

The key to this transformation is the integration of existing commercial equipment and sensors, leveraging available communication channels to facilitate seamless interactions. Importantly, these solutions are designed to cater to individuals who may not be tech-savvy, ensuring inclusivity. A noteworthy shift in the healthcare industry is the transition from hospital-centered applications to patient-centered ones. This patient-centric approach fosters more personalized and responsive healthcare services.

Advanced healthcare services, underpinned by IoT technologies, provide customers with valuable data streams and tailored advice for prevention and improving health conditions. This proactive approach empowers the elderly and their caregivers to lead better, more independent lives while effectively managing their health and well-being. Ultimately, these advancements in healthcare aim to enhance the quality of life for individuals facing health challenges in an aging society.

Transport The increasing demand for parking spaces in major cities is a significant challenge, contributing to heightened search activity, increased CO_2 emissions, traffic congestion, and worsened air pollution. Addressing these issues is crucial for the development of smart and sustainable urban environments. Two pivotal aspects of building smart cities are efficient parking management and optimized transportation systems. Cities are increasingly focusing on enhancing their parking infrastructure, leveraging IoT data to gather new insights and perform advanced analyses[28].

In the realm of transportation use cases, IoT-driven services employing prediction and optimization algorithms play a crucial role. These services can provide well-balanced parking plans that consider user preferences, historical parking data, real-time demand, and other relevant factors. This data-driven approach enables the emergence of innovative business models, such as dynamic pricing based on parking space attractiveness, hourly variations, or customer loyalty.

Furthermore, IoT technologies facilitate the fine-tuning of parking space management and proximity to entry points based on user-defined profiles. This customization caters to the specific needs of different residents and inhabitants. When optimizing parking resources, factors like safety, predictability, reliability, accessibility, and user comfort are taken into account. Additionally, access control and assessment systems are integrated into the overall solution. The type of user, whether a visitor or a resident, influences the suggested parking location, with careful consideration given to the needs of healthcare and emergency service organizations. This holistic approach to parking and transportation management contributes to improved urban mobility and the overall quality of life in cities.

Farming The agriculture sector is undergoing significant transformation due to the processes of industrialization and modernization, coupled with the challenges of declining agricultural land and increasing global population. In this context, the Internet of Things (IoT) is playing a pivotal role in revolutionizing agriculture by introducing advanced technologies that enhance production efficiency, improve the quality and quantity of agricultural products, and reduce production costs[29]. These IoT-enabled solutions are strategically applied throughout the entire agricultural production process to maximize the utilization of limited arable land.

The future of agriculture is marked by the integration of precision farming and smart farming practices. Precision farming, often referred to as precision agriculture, involves the use of IoT technologies to provide accurate supply and demand forecasts, real-time monitoring and resource management, and the maintenance of production quality throughout the entire life cycle of agricultural goods[30]. This approach allows farmers to precisely detect fluctuations in agricultural conditions and apply targeted techniques, such as optimizing the use of fertilizers and pesticides, to increase efficiency and reduce waste.

Furthermore, smart farming extends to livestock management, enabling farmers to monitor the real-time demands of each animal and provide appropriate treatment or feeding. This level of precision in animal care ensures the health and nutrition of the animals, contributing to the overall sustainability and productivity of the agriculture sector. In essence, IoT-driven farming practices are instrumental in addressing the complex challenges facing agriculture, from land scarcity to resource management and animal welfare, ultimately leading to more sustainable and efficient food production.

Challenges in IoT: Navigating the Complexities of a Transformative Technology

The barriers between our personal and professional lives have been broken down by the Internet of Things' widespread influence. As this game-changing technology develops further, it poses a variety of problems that need careful analysis and solutions. These difficulties are diverse, covering both technical complexities and a fast expanding range of non-technical factors, each of which is essential in determining the course of IoT.

From a technical perspective, ensuring the reliability and stability of IoT systems stands as a paramount challenge. This includes addressing issues of data security, privacy, and scalability as we grapple with the vast volumes of data generated by IoT devices. Achieving seamless interoperability among diverse devices and platforms presents another complex hurdle that necessitates the development of standardized protocols and data formats.

However, it is imperative to recognize that IoT's challenges extend far beyond the realm of technology. Non-technical facets, including ethical, legal, social, and economic dimensions, hold increasing significance. Ethical quandaries emerge concerning data collection, consent, and the responsible utilization of this wealth of information. Legal frameworks must be meticulously crafted to address complex issues of data ownership, liability, and regulatory compliance. Moreover, IoT's profound societal implications, such as its effects on employment dynamics and digital equity, warrant careful scrutiny and the formulation of informed policies.

Crucially, these technical and non-technical challenges are inherently interconnected. Technical obstacles can give rise to profound non-technical dilemmas, and societal expectations and legal frameworks adapt in response to the evolving IoT landscape. As IoT continues to infiltrate various aspects of our daily existence and industries, a holistic understanding of these intricate challenges is indispensable to unlock the full potential of this groundbreaking technology. This section embarks on a comprehensive exploration of these challenges, shedding light on the intricate interplay between technology and society in the IoT domain.

Technical challenges In the realm of Internet of Things (IoT), several profound technical challenges stand as formidable obstacles to realizing the full potential of this transformative technology. These challenges are multifaceted and multifarious, spanning various aspects of IoT implementation, from fundamental interoperability issues to the intricate web of security and privacy concerns. Delving into the technical intricacies of IoT, we unravel these pressing challenges.

Interoperability from technical up to semantic level Interoperability is the lifeblood of IoT, facilitating seamless communication between the vast array of devices and platforms that constitute this interconnected ecosystem. At its simplest, syntactic interoperability ensures that data exchanged between IoT entities adheres to standardized data structures and formats, mitigating the risk of misinterpretation. However, as IoT extends its reach to higher application layers, the challenge of semantic interoperability emerges. This entails extracting meaningful, context-rich information from the raw data churned out by IoT devices at the ”things” layer. The effective orchestration of services at the service layer hinges on this semantic comprehension. Additionally, the dearth of universally accepted standards further complicates interoperability, necessitating concerted efforts to harmonize disparate systems, subsystems, devices, and applications.

Security and Privacy The bedrock of any robust IoT system is its ability to provide airtight security measures. IoT systems must guarantee data confidentiality, integrity, and availability to meet stringent security requirements. Simultaneously, the vast amount of data collected by IoT devices, often including sensitive user information, such as personal identifiers and financial data, makes user privacy a paramount concern. The potential for malevolent actors to exploit vulnerabilities and misuse user data underscores the critical importance of privacy safeguards in IoT device development.

Connectivity Challenges: Centralised vs. Decentralised concepts IoT applications demand intricate decisions regarding architectural strategies, particularly in the context of data management and processing. A centralized approach, where a singular server handles all service implementations and data storage, may seem viable for small-scale deployments. However, scalability becomes a major impediment in scenarios involving billions of interconnected devices. Centralization introduces performance bottlenecks and becomes a single point of failure, undermining reliability and security. Conversely, decentralized paradigms, such as edge computing, aim to address these issues by pushing higher-level functions closer to the ”edge” devices. This mitigates the challenges of scalability and reliability but introduces a distinct set of difficulties. Concurrency problems, security concerns, bandwidth management, and data handling complexities all come to the forefront in decentralized models. Additionally, peer-to-peer connections, while promising local data control, introduce security challenges that require vigilant device owners and organizations to fend off potential breaches.

In navigating these technical challenges, the IoT community faces a multifaceted landscape that necessitates innovative solutions, industry-wide cooperation, and a holistic approach to ensure the continued growth and maturation of this transformative technology. This section delves deeper into these challenges, offering insights into their complexities and potential avenues for resolution.

Nontechnical challenges Beyond the intricate web of technical challenges, the realm of IoT is also riddled with nontechnical hurdles, each carrying its own weight in shaping the trajectory of IoT adoption and evolution. These challenges span a spectrum of issues that extend far beyond lines of code and technical specifications, delving into the realms of legality, privacy, user awareness, permissions, and asset maintenance.

1. **Legal and privacy issues**. IoT infrastructure, with its vast network of interconnected objects, is an intricate web that connects, identifies, communicates with, monitors, and regulates a myriad of entities, including users, sensors, devices, and appliances. The effective functioning of IoT systems often necessitates the collection of extensive data, paving the way for potential misuse and privacy violations. Incorrectly implemented data collection procedures can give rise to profound ethical and legal concerns, as the line between user convenience and data privacy becomes increasingly blurred. Additionally, the complex landscape of software licensing within IoT ecosystems can further compound policy challenges.

2. **Customer awareness**. A critical challenge in the IoT landscape revolves around the level of awareness among IoT device owners regarding the handling and utilization of their data. A significant portion of IoT consumers remains largely uninformed about how their data is harnessed. A McAfee study[31] found that a staggering 33% of respondents expressed concerns about their ability to monitor and manage the use of their data by businesses. This lack of awareness not only poses privacy and security risks but also has the potential to impede the widespread adoption of IoT technology.

3. **Customer permission**. The integration of IoT applications often necessitates the addition or replacement of sensors and devices within existing systems. Moreover, users may be required to modify their conventional behaviors to align with the functioning of IoT applications, such as the transformation of a traditional building into a smart one. This

transformation inherently involves the gathering, transmission, and analysis of personal data. Consequently, the process of persuading users to embrace and employ IoT applications hinges on providing transparent and easily accessible privacy regulations, ensuring that users have a clear understanding of how their data will be managed.

4. **Asset maintenance**. The operational viability of IoT systems relies heavily on the proper maintenance and upkeep of sensors and equipment. Unfortunately, the scarcity of adequately trained support personnel and challenges related to asset maintenance pose significant obstacles in this regard. Ensuring that IoT assets remain in optimal working condition is pivotal for the uninterrupted flow of data and the seamless functioning of IoT applications.

These nontechnical challenges underscore the multifaceted nature of IoT implementation and its far-reaching implications for society. Addressing these challenges necessitates a holistic approach that encompasses legal frameworks, user education, privacy regulations, and maintenance strategies to foster a conducive environment for the continued growth and ethical development of IoT technology.

2.5.4. Preliminary Conclusions: Navigating the IoT Landscape

The Internet of Things (IoT) has woven itself into the very fabric of our daily existence, becoming an integral part of our lives and exerting an ever-expanding influence on the world around us. As IoT pervades our homes, cities, industries, and beyond, it offers tangible benefits, transforming the way we interact with technology and the environment.

Illustrative use cases of IoT systems underscore the positive impact they bring to our daily routines and the surrounding ecosystem. From streamlined home automation to efficient energy management and innovative healthcare solutions, IoT exemplifies its potential to enhance convenience, sustainability, and well-being.

Yet, the road to realizing the full potential of IoT is not without its hurdles. IoT initiatives encounter multifaceted challenges, spanning both the technical realm and the complex landscape of stakeholders. Technical challenges encompass issues like interoperability, security, and connectivity, demanding innovative solutions and standardization efforts.

Equally critical are the nontechnical challenges emanating from the very users and entities that IoT intends to serve. Clear communication and education are imperative, as businesses must navigate the intricate terrain of stakeholder engagement. End users, representing a diverse spectrum of society, must be made cognizant of the benefits and risks associated with IoT.

For instance, a startling statistic from a 2018 McAfee study[31] reveals that fewer than 40% of individuals employ adequate identity protection measures. The unawareness of IoT's underlying processes may overshadow its myriad benefits, leading to apprehensions about security and data privacy.

In essence, the success of IoT hinges on a harmonious interplay between technology and society. Businesses crafting IoT systems must engage with users transparently and effectively, illuminating the transformative potential while addressing concerns. By cultivating awareness, fostering trust, and implementing robust safeguards, the IoT landscape can evolve into a force that not only augments our lives but does so responsibly and ethically.

2.6. IoT Platforms: Orchestrating the IoT Ecosystem

In the dynamic landscape of the Internet of Things (IoT), an array of platforms has emerged to expedite and refine the implementation of IoT projects. These platforms constitute a pivotal element within the intricate web of physical and virtual entities, offering a suite of tools to simplify the intricacies of IoT endeavors.

This chapter embarks on a comprehensive exploration of IoT platforms, spanning the entire spectrum from the gateway level to the vast expanses of the cloud. It delves into the architectural underpinnings of these platforms and elucidates the process of crafting a bespoke IoT platform. Please note that certain portions of the content and findings presented in this chapter have been previously disseminated in [2] and [32], and have been thoughtfully incorporated into this dissertation to provide a comprehensive and coherent perspective on the subject matter.

At its core, an IoT platform is a multi-layered technological construct, meticulously engineered to furnish an assortment of readily deployable functionalities, thus catalyzing the realization of IoT initiatives. The linchpin of IoT platforms lies in their innate capacity to facilitate seamless communication among the constellation of interconnected entities.

The sprawling IoT architecture is an intricate mosaic, composed of myriad components. As delineated in Figure 2.4, a holistic IoT ecosystem typically comprises five cardinal components: hardware, acting as the bedrock with sensors and devices; gateways serving as intermediaries; cloud-based data processing; communication protocols for connectivity; and user interfaces for interaction. Devoid of IoT platforms, a conspicuous chasm would loom between the hardware and application strata, impeding the synthesis of this multifarious ensemble.

The pivotal role of IoT platforms comes to the fore in bridging this divide and orchestrating a symphony of harmonious interactions. These platforms empower developers to navigate the labyrinthine realm of diverse hardware and software communication protocols, endow user devices with robust security measures, and orchestrate the orchestration of sensor data, encompassing collection, visualization, and analysis.

This chapter embarks on an illuminating journey through the facets of IoT platforms. It commences by elucidating the vantage point from which one can perceive the external environment. Subsequently, it delves into the contours of cloud platforms, IoT middleware, and IoT gateways, encompassing both hardware and software facets. The chapter also unravels the intricate skein

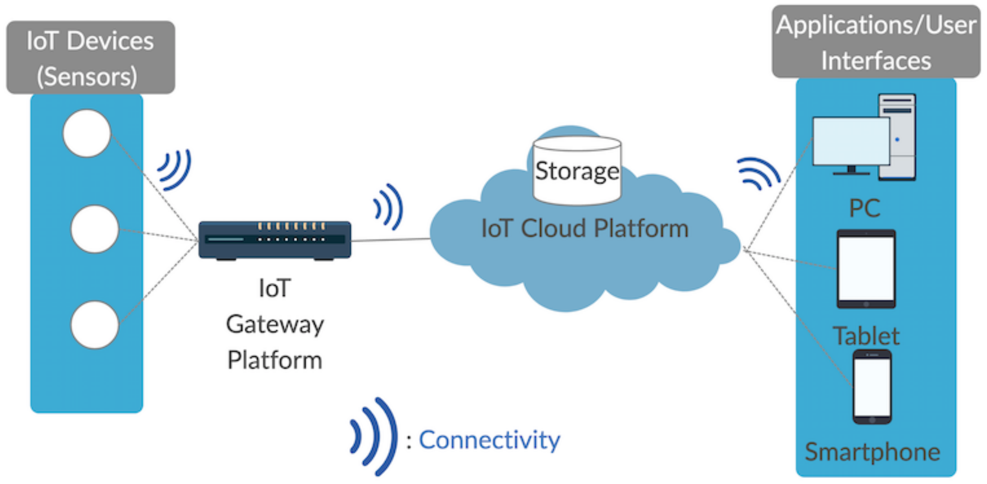


Figure 2.4.: *Place of IoT gateway and cloud platform in IoT Architecture*

of protocols, threading the IoT tapestry and interconnecting devices. It traverses the terrain of data processing and storage, illuminating the pathways to harness the torrent of data generated by IoT ecosystems.

The voyage concludes with an introduction to the design paradigm of linked appliances, an end-to-end journey that commences with the instantiation of electronics and culminates in the fine-tuning of firmware. Join us on this expedition as we unravel the multifaceted tapestry of IoT platforms and their pivotal role in sculpting the IoT landscape.

2.6.1. Sensing the real world: The Vital Role of Sensors

In the realm of the Internet of Things (IoT), sensors emerge as the quintessential tools that facilitate the acquisition of real-world inputs, thereby enabling interaction with the environment. These unassuming devices possess the remarkable ability to transmute measurements of diverse physical parameters, such as temperature, humidity, light, motion, heat, sound, and more, into discernible signals interpretable by humans.

In the ever-expanding domain of IoT, a diverse array of sensor types finds their way into an increasingly ubiquitous range of everyday objects. Within these smart devices, exemplified by the modern refrigerator (see Fig. 2.5), sensors assume the role of diligent sentinels, ceaselessly observing the environment and dutifully recording data pertaining to events or changes. This data spans a rich spectrum, encompassing metrics like ambient temperature, humidity levels, photographic imagery, and even video recordings. This continuous influx of information serves as the lifeblood that nourishes our ability to discern patterns, unveil trends, and craft insightful predictions.

To elucidate this symbiotic relationship between sensors and IoT applications, let us consider the hypothetical scenario of a smart refrigerator, replete with an assortment of sensors, as outlined in Fig. 2.6[34]. Here, we dissect the system architecture into its three fundamental components:

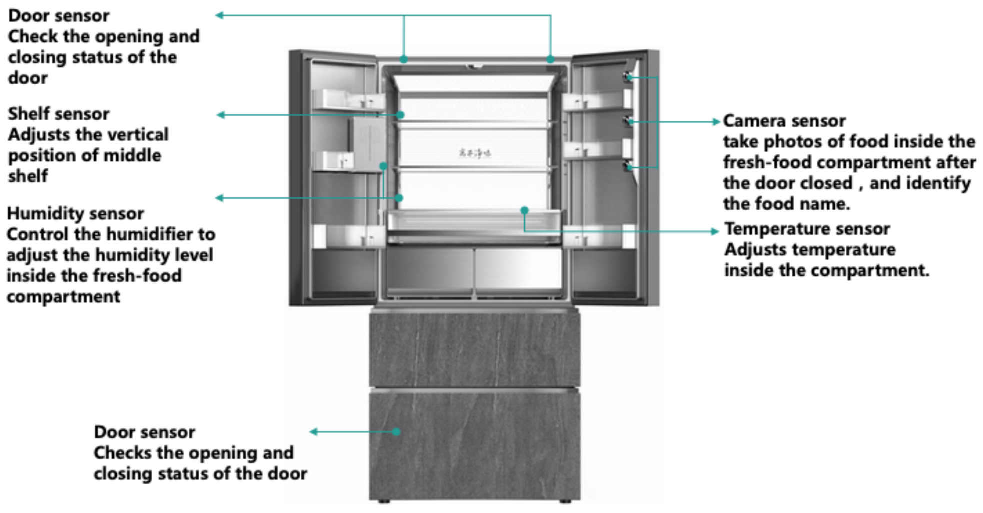

Door sensor
Check the opening and closing status of the door
Shelf sensor
Adjusts the vertical position of middle shelf
Humidity sensor
Control the humidifier to adjust the humidity level inside the fresh-food compartment
Door sensor
Checks the opening and closing status of the door
Camera sensor
take photos of food inside the fresh-food compartment after the door closed , and identify the food name.
Temperature sensor
Adjusts temperature inside the compartment.

Figure 2.5.: *Hisense Refrigerator's key sensors [33]*

- **Sensor-Equipped Container**: At the bedrock of this architecture lies the hardware domain. It comprises an ensemble of vital components, including Arduino and Raspberry Pi, which orchestrate the monitoring of the smart refrigerator's condition. Armed with an array of sensors - capable of measuring temperature, humidity, and luminosity - this hardware ecosystem stands poised to scrutinize the refrigerator's environment. It culminates its mission by capturing visual data, encapsulating an instant snapshot of the surroundings at the behest of temperature and humidity sensors.

- **Information Server**: Functioning as the neural hub of the system, the Information Server serves as the vital link between the user interface, application layer, and the sensing hardware. With an intricate web of interconnectivity, it processes incoming sensor signals, executing a gamut of operations in response.

- **Maintenance Application**: Endowing the user with omnipotent control, the Maintenance Application takes the form of a mobile application, nestling comfortably on smartphones or tablets. This interface provides users with unfettered access to the troves of processed data residing on the server. Even when physically distant from their abode, users can instantaneously tap into the repository of their connected refrigerator's insights.

In essence, sensors represent the pivotal conduit through which the IoT cosmos interacts with the tangible world. These unassuming devices, armed with the power to translate physical phenomena into digital signals, form the foundational bedrock upon which the edifice of IoT is erected, ushering forth an era where inanimate objects resonate with life and intelligence.

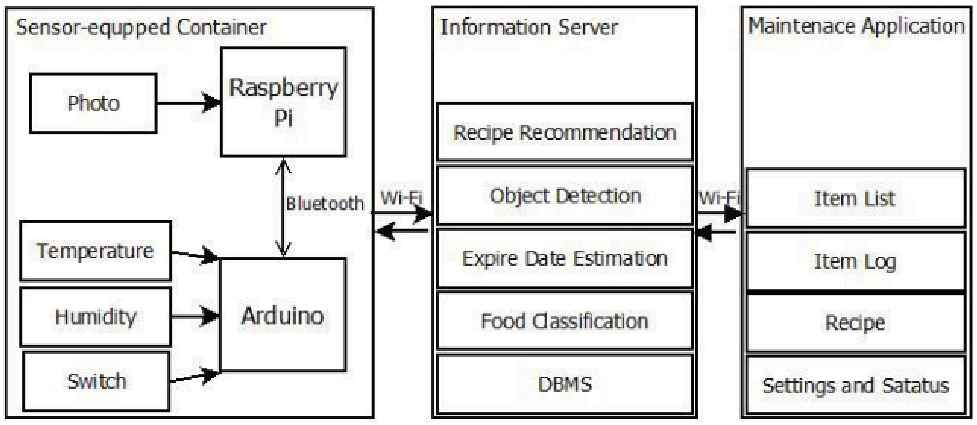


Figure 2.6.: *System Architecture of Smart Refrigerator [34]*

2.6.2. IoT gateway platforms: Bridging the Physical and Digital Realms

In the intricate tapestry of the Internet of Things (IoT), the notion of a gateway emerges as a pivotal linchpin, facilitating seamless communication between distinct realms. In essence, an IoT gateway, whether manifested in hardware or software, plays the venerated role of a bridge, forging the crucial link between IoT devices and the boundless expanse of the cloud.

The crux of the matter lies in the elemental flow of data and information within the IoT ecosystem. IoT devices, each an autonomous data generator in its own right, form the bedrock upon which the edifice of this technological marvel is constructed. These devices, endowed with diverse low-level communication protocols, embark on a ceaseless journey of observation and measurement, their purpose being to translate real-world phenomena into digital signals.

However, the digital realm of the cloud, wherein the fruits of these devices' labor are destined to be harvested, stands distant and disparate. Enter the IoT gateway, an intrepid sentinel perched at the interface of these disparate worlds.

At its core, an IoT gateway serves as a nexus, a veritable crossroads where data streams converge and diverge. It excels in the art of translation, deftly converting the cryptic dialects of diverse low-level protocols into a common tongue, one universally understood within the realm of the cloud. In essence, it bridges the chasm between the tangible and the digital, facilitating the fluid transfer of data and information.

To construct these vital bridges, the IoT landscape boasts a profusion of hardware and software platforms, each offering unique capabilities and features. In the ensuing subsections, we delve deep into the annals of these IoT gateway platforms, unraveling their intricacies and elucidating their nuances.

Evaluation of Gateway Hardware Platforms: Bridging the Physical Divide

In the realm of IoT gateways, a panoply of hardware platforms emerges as the bedrock upon which IoT systems are built. These tangible gateways, equipped with microcontrollers and CPUs, serve as the conduits that bridge the physical and digital worlds. This subsection embarks on an exploratory journey, shedding light on a few selected hardware gateways that exemplify the diversity within this domain.

Raspberry Pi 3 Model B This gateway, adorned with the BCM43438 chip, boasts WiFi 802.11n connectivity and Bluetooth Classic 4.1 capabilities. It features a 40-pin extension header equipped with UART, I2C, and SPI connectors, facilitating seamless hardware connections. Four USB host ports further enhance its versatility.

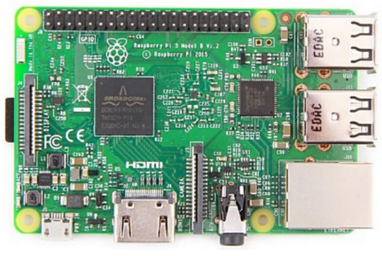


Figure 2.7.: *Front side of Raspberry Pi 3 Model B board [35]*

Banana Pro The Banana Pro, housing the AP6181 chip, forgoes Bluetooth and ZigBee modules in favor of WiFi 802.11n connectivity. Its 40-pin expansion header accommodates three UART devices, two I2C buses, and one SPI bus, while pins 16 and 17 facilitate Controller Area Network (CAN) connections in industrial and automotive applications. This gateway offers one USB On-The-Go (OTG) port and two independent Universal Serial Bus (USB) host ports. Fig.2.8 displays the banana pi pro front side.

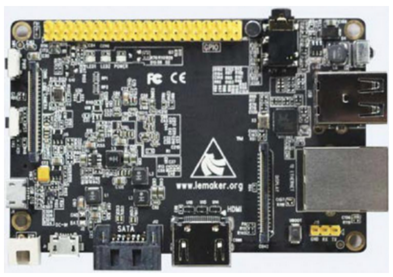


Figure 2.8.: *Front side of Banana Pro board [36]*

Pine A64+ The Pine A64+ single-board computer caters to a spectrum of Android OS versions. With two expansion headers, the first featuring 40 pins and the second known as the Euler bus, sporting 34 pins, it offers ample connectivity options. Realtek RTL8723BS powers its wireless module, enabling WiFi 802.11n and Bluetooth Classic 4.0 functionalities.

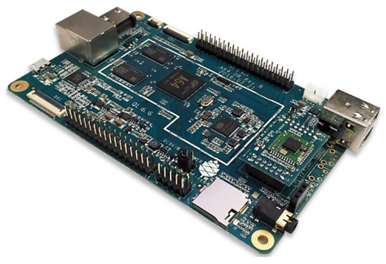


Figure 2.9.: *Front side of Pine A64+ board [37]*

Cubietruck This gateway leverages an AP6210 wireless chip to deliver WiFi 802.11n and Bluetooth Classic 4.0 support. Its two expansion headers, replete with 54 pins, foster connections for three UART devices, one I2C bus, and an SPI bus. It even accommodates modules like the Core2530 ZigBee module via the DVK570 expansion board. Two USB host ports and one USB OTG port enhance its connectivity.

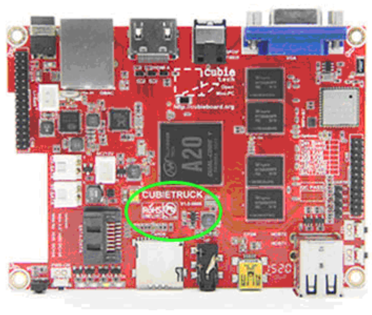


Figure 2.10.: *Front side of Cubietruck board [38]*

Intel Edison Intel's Edison module emerges as a versatile system-on-a-chip, facilitating the creation of IoT and wearable devices. WiFi and Bluetooth 4.0 LE capabilities empower this gateway to bridge the physical and digital realms seamlessly.

ESP8266 Hailing from Espressif Systems, the ESP8266[1] stands as a testament to low-power Wi-Fi microprocessors. With a complete TCP/IP stack and microcontroller capabilities, it offers a compact yet potent solution for IoT gateways.

[1]ESP8266: `http://https://www.espressif.com/en/products/hardware/esp8266ex/overview`

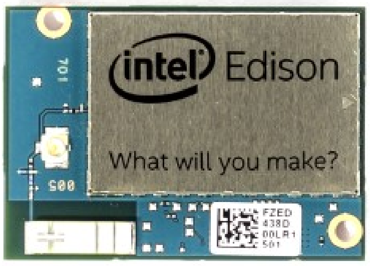


Figure 2.11.: *Front side of Intel Edison board [39]*

Arduino Uno WiFi A stalwart in the realm of microcontroller boards, the Arduino Uno WiFi[2] combines the prowess of the ATmega328P with WiFi capabilities. It boasts an inbuilt Inertial Measurement Unit (IMU) and a secure ECC608 crypto chip accelerator, enhancing its connectivity and security features.

BeagleBoard BeagleBoard's cheap single-board computers[3], based on Texas Instruments processors with ARM Cortex-A series cores, offer open-source blueprints and readily available parts, fostering compatibility and flexibility in hardware creation.

SODAQ The SODAQ ecosystem[4] offers a range of hardware products, such as Autonomo, SODAQ SARA AFF N211, and Mbili. These boards are engineered for low power consumption, capable of running on small lithium batteries and solar panels, making them ideal for remote and energy-efficient IoT applications.

In this intricate web of hardware gateways, each platform embodies a unique blend of features and capabilities, catering to the diverse needs of IoT systems. These gateways stand as the tangible foundations upon which the ethereal IoT ecosystem is built.

Evaluation of Gateway Software Platforms: Navigating the Digital Realm

In the realm of IoT gateways, where software prowess meets hardware might, an array of open-source software platforms plays a pivotal role in building the bridge between the physical and digital worlds. This subsection explores these digital gatekeepers that orchestrate the flow of data in IoT systems.

OpenHAB plaftorm OpenHAB, an open-source platform rooted in Java, is renowned for simplifying home automation. However, its versatility extends to IoT gateways. Unlike cloud-dependent solutions, OpenHAB directly communicates with local devices and retains data on-site. It operates through

[2]Arduino Uno WiFi: `http://store.arduino.cc/arduino-uno-wiFi-rev2`
[3]BeagleBoard: `https://beagleboard.org/beagleboard/`
[4]SODAQ: `https://shop.sodaq.com/sodaq-sara-aff-r410m.html`

Equinox runtime, executing bundles based on the OSGi architecture. Fig. 2.12 provides a glimpse of its architecture.

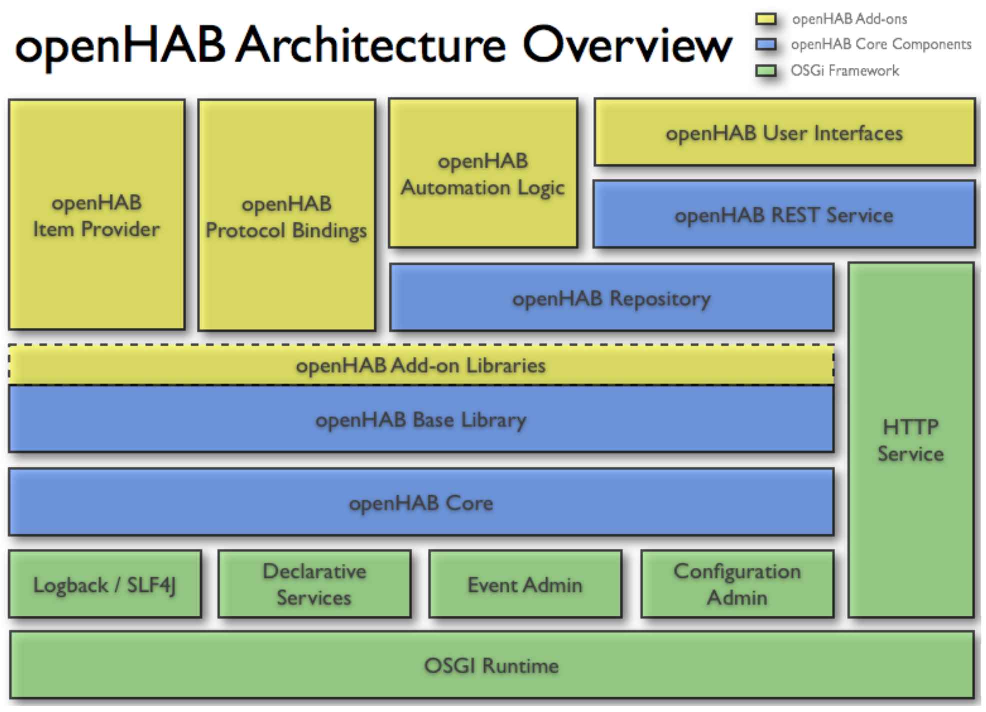


Figure 2.12.: *OpenHAB architecture [40]*

DeviceHive platform DeviceHive emerges as an open-source M2M communication platform that encompasses IoT gateway capabilities. Deviating from the OSGi path, it adopts a D-BUS-based design, as illustrated in Fig. 2.13. DeviceHive serves as a scalable, hardware-centric, and cloud-agnostic microservices platform, facilitating device management, connectivity configuration, control, and behavior analysis via APIs supporting various protocols.

OpenRemote platform Born in 2009 to resolve integration issues among diverse M2M communication protocols, OpenRemote amalgamates multiple protocols and technologies for smart city and building automation, aided by visualizations. Its architecture, depicted in Fig. 2.14, comprises three key components: local runtime controller, control panels (apps), and cloud-based configuration tools named OpenRemote Designer.

AllJoyn platform Introduced in 2013, AllJoyn is an open-source framework designed to ensure seamless device interoperability. This framework provides an abstraction layer compatible with Android, iOS, Linux, and Windows, facilitating application development without tethering to a specific operating system[40]. AllJoyn encourages proximity networking and adaptability, as outlined in Fig. 2.15.

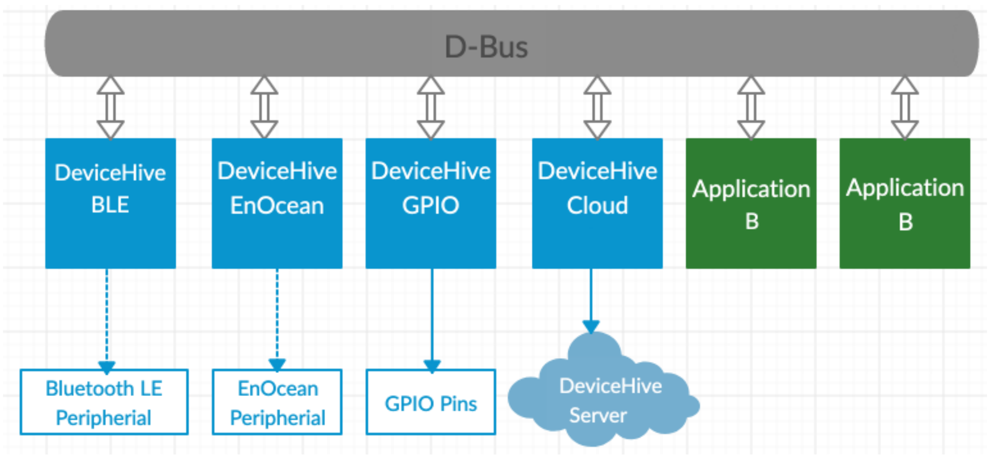


Figure 2.13.: *DeviceHive architecture overview [41]*

IoTivity platform Debuting in 2015, IoTivity ushers in a new standard for interconnecting wired and wireless devices in the IoT landscape. Its architecture, represented in Fig. 2.16, offers an efficient framework suitable for smart devices, aiming to address the burgeoning needs of IoT through device-to-device connectivity[43].

Eclipse Kura platform Eclipse Kura, a Java-based framework harnessing OSGi, empowers the creation of IoT gateways. Operating atop the Java Virtual Machine (JVM), Kura simplifies software development through reusable building blocks. It streamlines network configuration, IoT server connectivity, and remote gateway management. Kura's APIs grant access to hardware gateway interfaces like GPIOs, I2C, Serial ports, and more, facilitating hardware-level interactions.

These software platforms, akin to the conductors of a digital orchestra, harmonize the diverse elements within IoT gateways, orchestrating the seamless flow of data between the physical and digital realms.

2.6.3. IoT Middleware

IoT middleware emerges as the linchpin, connecting and accelerating various facets of IoT systems, from resource identification to data management, knowledge extraction, privacy, and security enhancement. This subsection casts a spotlight on noteworthy IoT Middleware Frameworks.

Hydra The Hydra project[5], a Networked Embedded System middleware, sets the stage for interoperability and security in IoT applications via a P2P network. It empowers developers to seamlessly integrate diverse physical objects into their applications, offering straightforward web service interfaces. Regardless of the underlying network technologies (e.g., Zwave, WiFi, ZigBee,

[5]Hydra Project: `https://vicinity2020.eu/vicinity/content/hydra`

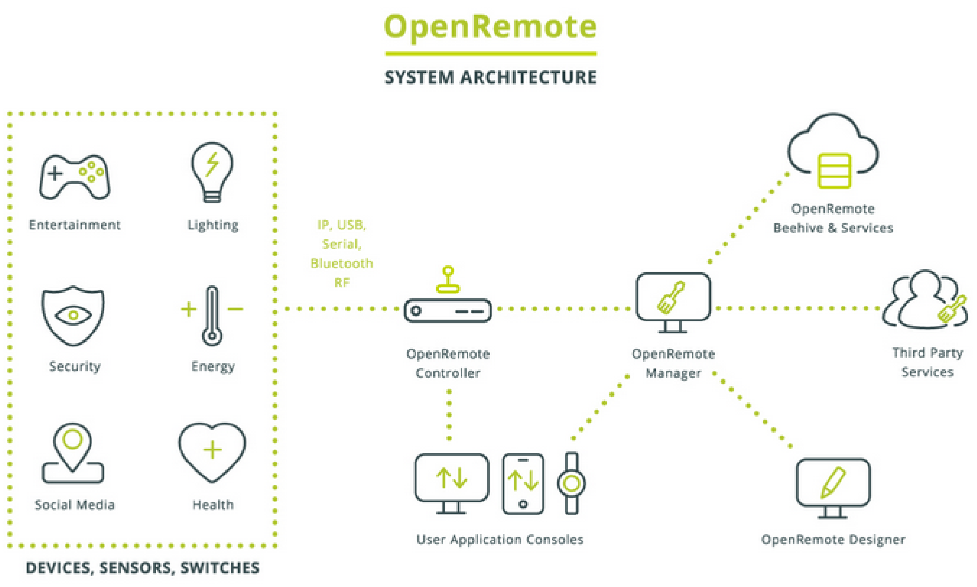


Figure 2.14.: *OpenRemote architecture [40]*

LoRaWAN), Hydra excels in unifying heterogeneous physical objects. Armed with P2P communication tools, diagnostics, Semantic Model Driven Architecture, and device/service discovery capabilities, Hydra ensures the safety and reliability of objects and services. Notably, it transforms into "LinkSmart"[6] as the Hydra project concludes.

Ubiware Ubiware adopts an agent-based middleware approach, endowing each resource with a proactive agent. It boasts a three-layer agent architecture comprising the behavior engine layer, intermediate layer, and resources layer[45]. Drawing inspiration from ubiquitous computing, it integrates distributed AI, semantic web, and human-centric computing technologies. Ubiware's domain-independent nature facilitates the interconnection, interoperability, communication, interaction, self-awareness, and planning of various resources, systems, and devices[46]. Metadata and ontologies form its core components, crucial for ensuring interoperability among objects.

OpenIoT OpenIoT[7], a versatile middleware infrastructure, enables flexible configuration and deployment of algorithms for collecting, filtering, and processing information from internet-connected components. Embracing a utility cloud computing delivery paradigm, OpenIoT paves the way for extensive smart IoT applications at scale. It facilitates access to additional IoT-based resources and technology, supporting the creation and management of IoT resource ecosystems. OpenIoT also offers on-demand utility IoT services such as sensing as a service, impacting various scientific and technological fields,

[6]LinkSmart: `https://www.linksmart.dk/news.php`
[7]OpenIoT: `https://github.com/OpenIotOrg/openiot`

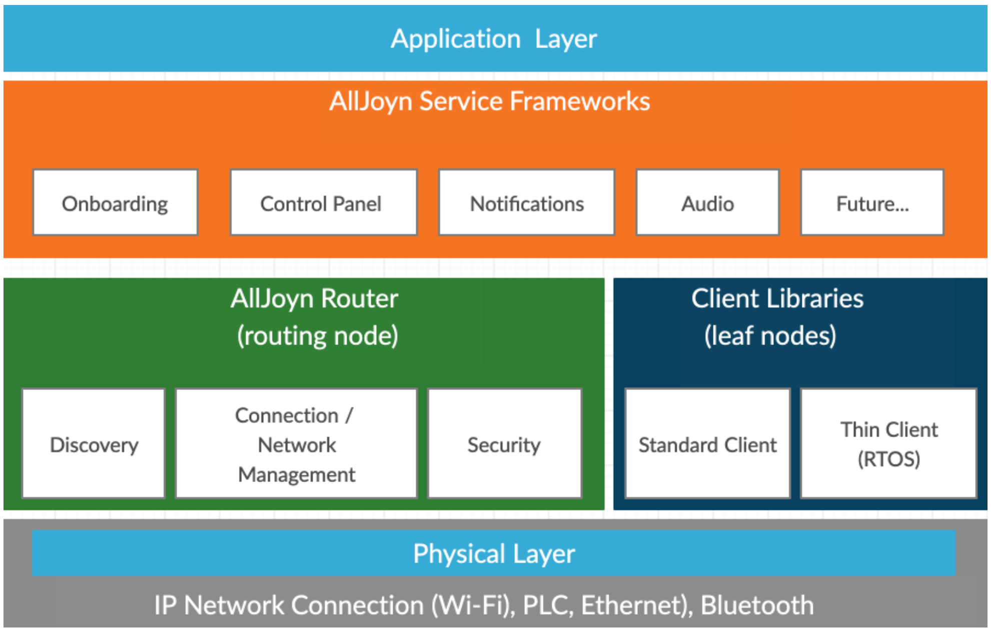

Application Layer
AllJoyn Service Frameworks
Onboarding
Control Panel
Notifications
Audio
Future...
AllJoyn Router (routing node)
Discovery
Connection / Network Management
Security
Client Libraries (leaf nodes)
Standard Client
Thin Client (RTOS)
Physical Layer
IP Network Connection (Wi-Fi), PLC, Ethernet), Bluetooth

Figure 2.15.: *AllJoyn architecture [42]*

including middleware for sensors and sensor networks, ontologies, semantic models, annotations, and cloud/utility computing.

FIWARE The FI-PPP program leverages FIWARE[8], an open, architectural, and operational software, to foster the generation and delivery of services across diverse domains. FIWARE envisions an open and sustainable ecosystem centered on open, royalty-free, and implementation-driven software platform standards. It fuels the creation of innovative, intelligent applications across a multitude of fields[40]. The FIWARE community, including the FIWARE Foundation and FIWARE OSC (Open Source Community), actively supports and drives the FIWARE platform's evolution.

These IoT middleware frameworks function as digital bridge builders, facilitating seamless connectivity and acceleration across IoT landscapes, ultimately enhancing interoperability, security, and innovation.

2.6.4. IoT cloud platforms

IoT's convergence with cloud computing opens the door to myriad possibilities. IoT cloud platforms, such as those offered by VICINITY, Amazon Web Services (AWS), Microsoft Azure, and Google Cloud, provide the critical infrastructure needed to support IoT applications.

VICINITY platform Launched under the Horizon 2020 Research and Innovation Program, the VICINITY platform weaves a decentralized, open virtual neighborhood network[47]. It acts as an ”interoperability as a service”

[8]FIWARE: `https://www.fiware.org/`

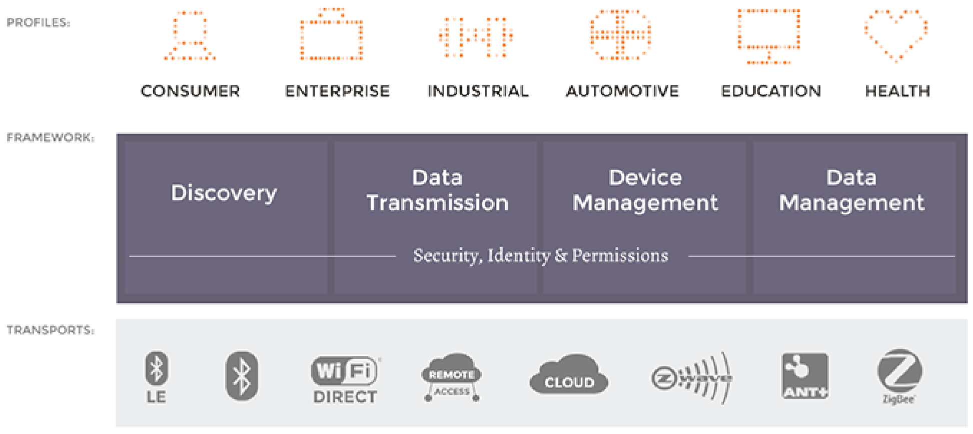


Figure 2.16.: *IoTivity architecture [43]*

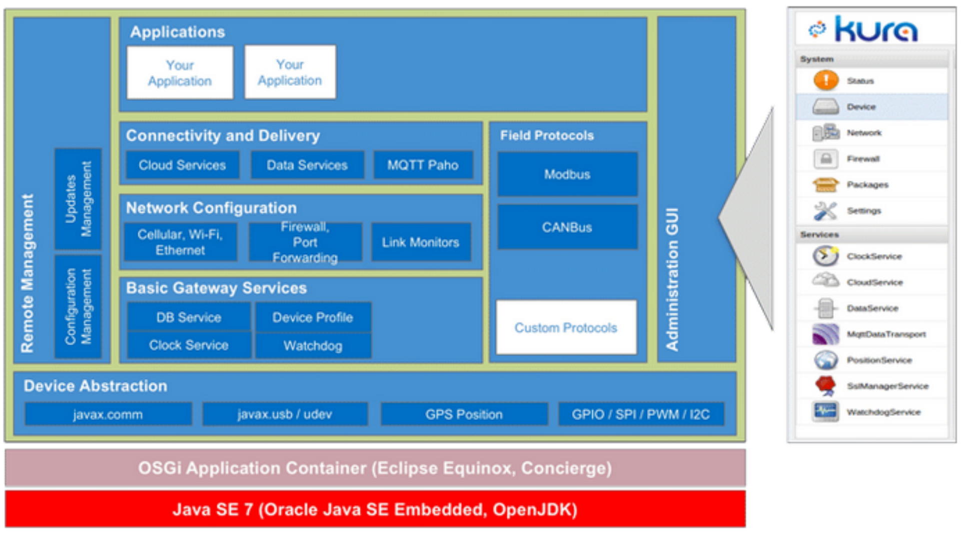


Figure 2.17.: *Eclipse Kura overview [44]*

provider, akin to a social network for IoT. This platform forges connections across diverse IoT ecosystems, enabling users to interact with smart devices as if they were part of their own ecosystem. VICINITY's scope spans smart home, smart energy, smart transportation, and eHealth applications.

Amazon Web Services platform AWS, a trailblazer in cloud computing since 2006, extends its prowess to IoT. Supporting device-to-device and device-to-cloud connections, AWS bolsters various communication protocols like WebSockets, MQTT, and HTTP. Security and data protection are paramount, featuring robust authentication and encryption. Key components encompass AWS IoT Device SDK, Device Gateway, Authentication and Authorization, Registry, Device Shadows, and Rules Engine, as shown in Fig.2.19[48].

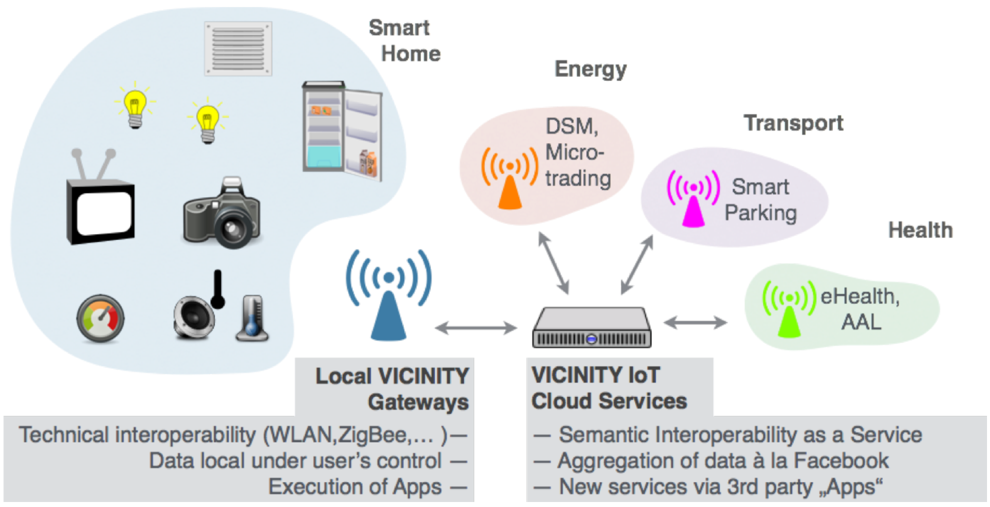

Smart Home
Energy
DSM, Micro-trading
Transport
Smart Parking
Health
eHealth, AAL
Local VICINITY Gateways
VICINITY IoT Cloud Services
Technical interoperability (WLAN,ZigBee,...)—
Data local under user's control —
Execution of Apps —
— Semantic Interoperability as a Service
— Aggregation of data à la Facebook
— New services via 3rd party „Apps"

Figure 2.18.: *VICINITY Overview [40]*

Microsoft Azure platform Launched in 2010, Microsoft Azure offers a comprehensive cloud computing platform. Azure encompasses Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Its multifaceted features include scalability, data analytics, high availability, privacy, security, and a pay-as-you-go model (refer to Figure 2.20).

Google Cloud platform Born in 2008, Google Cloud Platform leverages Google's infrastructure to empower users in creating, launching, and expanding apps, websites, and services. GCP's data lifecycle spans acquisition, processing, storage, and visualization, with real-time stream processing capabilities (refer to Figure 2.21). Its key features encompass scalability, high performance, security, compliance, and an environmentally friendly cloud environment.

While these IoT cloud platforms offer similar capabilities, their distinctive qualities ensure they remain influential players in the IoT-cloud landscape. Whether it's VICINITY's focus on interoperability, AWS's robust security, Azure's comprehensive cloud services, or GCP's performance and sustainability, each platform contributes to the ever-evolving IoT ecosystem.

2.6.5. Selection of IoT platforms for building own smart devices

In the dynamic world of IoT, choosing the perfect platform to build your smart devices is crucial. This section explores the factors to consider when selecting an IoT platform, helping you make informed decisions on your IoT journey.

Selection of IoT platform

In the intricate realm of the Internet of Things (IoT), each platform offers unique features and capabilities. Consequently, the task of selecting the most suitable IoT platform becomes a complex endeavor. This section provides

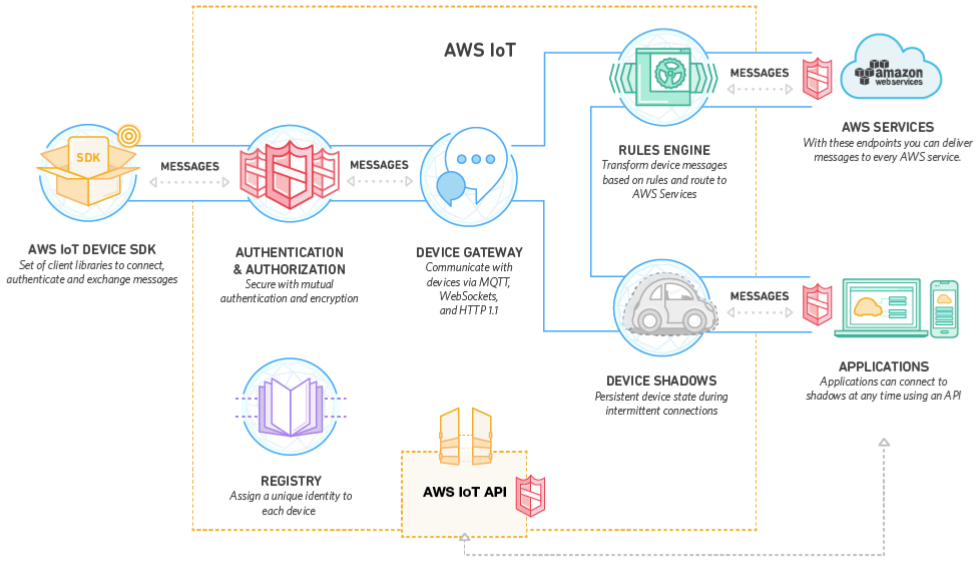

AWS IoT
SDK
MESSAGES
MESSAGES
MESSAGES
MESSAGES
AWS IoT DEVICE SDK
Set of client libraries to connect, authenticate and exchange messages
AUTHENTICATION & AUTHORIZATION
Secure with mutual authentication and encryption
DEVICE GATEWAY
Communicate with devices via MQTT, WebSockets, and HTTP 1.1
RULES ENGINE
Transform device messages based on rules and route to AWS Services
amazon web services
AWS SERVICES
With these endpoints you can deliver messages to every AWS service.
DEVICE SHADOWS
Persistent device state during intermittent connections
APPLICATIONS
Applications can connect to shadows at any time using an API
REGISTRY
Assign a unique identity to each device
AWS IoT API

Figure 2.19.: *Amazon Web Services Overview [49]*

guidance for making informed choices when seeking the optimal IoT platform for a project. The following steps are recommended:

1. Thorough Feature Comparison: Initiate the selection process by conducting a comprehensive analysis of various IoT platforms. Scrutinize their features, functionalities, and technical specifications.
2. Precise Smart Device Specification: Define precise specifications for the smart devices that are integral to your project. This includes delineating performance expectations and connectivity requirements.

In addition to the above steps, several key criteria should be considered in the decision-making process:

- **Hardware Platform Selection**: Evaluate IoT gateway hardware platforms based on criteria such as performance, connectivity capabilities, and the capacity for autonomous operation.
- **Software Platform Selection**: When selecting a software platform, examine aspects such as available IoT data gathering protocols, security features, portability, extensibility, developer documentation, availability of developer examples, and the robustness of the support community.
- **Cloud Platform Selection**: For IoT cloud platform selection, prioritize the following criteria:
  - **Robust Protocol Support**: Ensure the chosen platform offers robust support for data ingestion protocols.
  - **Offline Functionality**: Verify the platform's capability for reliable offline operation.

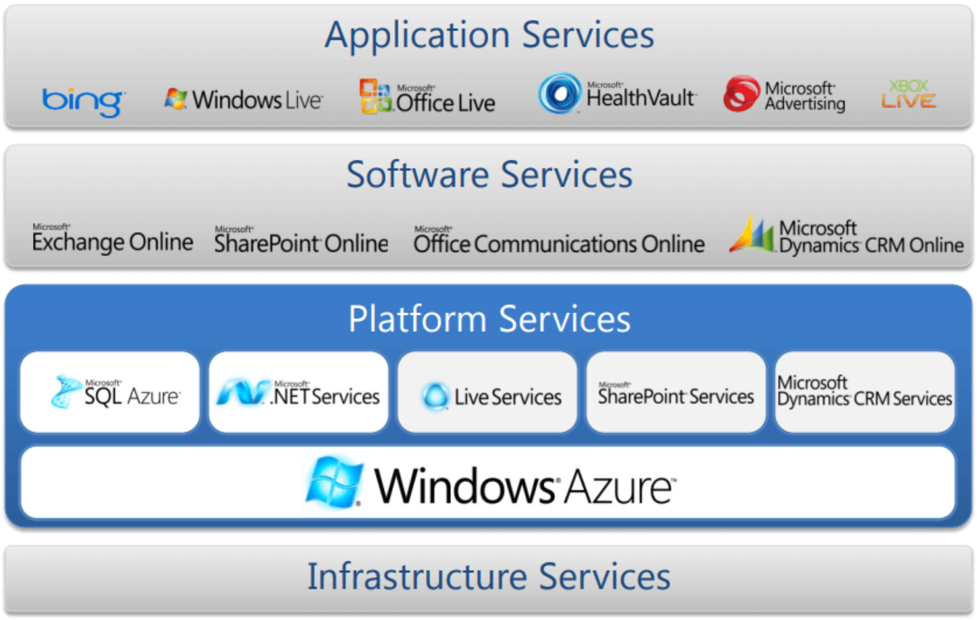


Figure 2.20.: *Microsoft Azure Overview [50]*

- **Device Lifecycle Management**: Confirm that the platform supports device lifecycle management through cloud-based orchestration.
- **Scalable Design**: Opt for a platform with a scalable architecture that remains independent of underlying hardware.
- **Analytics and Visualization**: Assess whether the platform provides comprehensive analytics and visualization tools, which can significantly impact data interpretation and decision-making.

By adhering to these systematic considerations and criteria, you can make a well-informed selection of an IoT platform tailored to the specific requirements of your project, ensuring its successful implementation in the realm of the Internet of Things.

2.6.6. Connecting devices

In the vast landscape of the Internet of Things (IoT), where the convergence of devices and data defines the ecosystem, the significance of connectivity cannot be overstated. It is through seamless communication protocols that intelligent entities, ranging from sensors to gateways, facilitate the transmission of data within the IoT framework. This section delves into the diverse array of connectivity options, encompassing both wired and wireless communication protocols, that underpin the IoT infrastructure.

Intricately woven into the fabric of IoT, these protocols are instrumental in orchestrating the transfer of data between the cloud and IoT endpoints. Each protocol carries its own set of features and advantages, tailored to specific

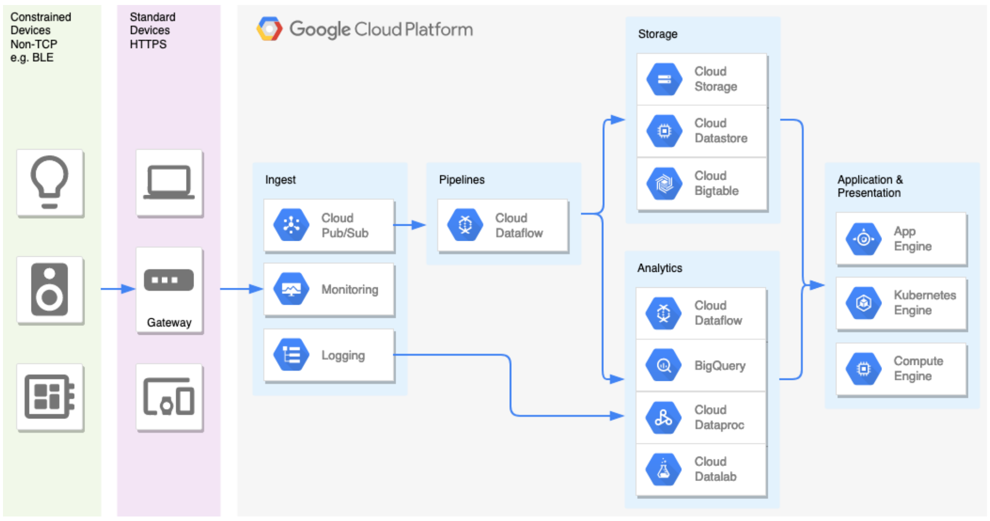


Figure 2.21.: *Google Cloud platform architecture [51]*

use cases within the IoT landscape. The ensuing paragraphs shed light on a selection of these IoT protocols, elucidating their functionalities and contextual relevance.

Device connectivity protocols

A panoramic view of IoT's technological landscape necessitates a meticulous exploration of the diverse protocols governing connectivity between IoT devices and applications. The Internet of Things (IoT) is a multifaceted realm that encompasses a spectrum of businesses, spanning from individual devices to extensive cross-platform deployments, amalgamating embedded technology, cloud systems, and real-time communication.

To bring coherence to this intricate web of IoT protocols, it is prudent to categorize them into distinct layers rather than shoehorning them into the traditional OSI model. This stratification provides a structured framework, ensuring a better comprehension of the IoT protocol landscape. Here are the pivotal layers for IoT protocols:

1. **Infrastructure Layer**: This layer encompasses protocols such as 6LowPAN, IPv4/IPv6, and RPL, which serve as the bedrock for IoT networks, managing the fundamental infrastructure on which IoT systems rely.

2. **Identification Layer**: In the realm of IoT, precise identification is paramount. Protocols like EPC, uCode, IPv6, and URIs facilitate the unique identification of IoT entities, ensuring seamless communication.

3. **Communication / Transport Layer**: The crux of IoT connectivity lies in this layer, with protocols like Wi-Fi, Bluetooth, and LPWAN facilitating the wireless transmission of data between devices and networks.

4. **Discovery Layer**: Discovery protocols like Physical Web, mDNS, and DNS-SD play a pivotal role in locating and identifying IoT devices within networks, a crucial aspect of IoT functionality.

5. **Data Protocols**: At the heart of IoT communication are data protocols like MQTT, CoAP, AMQP, Websocket, and Node. These protocols govern how data is formatted, transmitted, and received across IoT systems.

6. **Device Management Layer**: Device management is indispensable for IoT scalability and maintenance. Protocols like TR-069 and OMA-DM offer standardized mechanisms for overseeing and controlling IoT devices.

7. **Semantic Layer**: In the quest to make sense of IoT data, semantic protocols like JSON-LD and Web Thing Model provide a framework for structuring and interpreting IoT data in a meaningful way.

8. **Multi-layer Frameworks**: AllJoyn, IoTivity, Weave, and Homekit represent multi-layer frameworks that provide comprehensive solutions for IoT interoperability, bridging various aspects of the IoT stack.

In dissecting the IoT protocol landscape through this layered approach, a clearer understanding of the intricate web of IoT technologies emerges, paving the way for more informed decision-making in IoT design and implementation.

Generic protocols - WiFi, Cellular, Ethernet In the expansive landscape of IoT device connectivity, a trio of generic protocols — WiFi, Cellular, and Ethernet — emerges as the stalwarts of modern networking standards. These protocols, rooted in the comprehensive IEEE 802 Standard, serve as the foundational building blocks for a diverse range of IoT applications, spanning from the local area to wide area networks.

IEEE 802 Standard The IEEE 802 Standard is a comprehensive framework encompassing a multitude of networking specifications, intricately detailing both the physical and data-link aspects of network communication. Among the myriad components within this standard, four prominent specifications significantly influence IoT connectivity:

- 802.3 Ethernet: This specification defines the Ethernet protocol, a ubiquitous technology underpinning local area networks (LANs). Ethernet is celebrated for its reliability and high-speed data transmission capabilities.

- 802.11 Wi-Fi: Wi-Fi, characterized by various sub-specifications within the 802.11 family, revolutionized wireless connectivity. It employs the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol for efficient channel sharing, enabling seamless wireless communication.

- 802.15 Bluetooth/ZigBee: The 802.15 specification encompasses Bluetooth and ZigBee, two prominent wireless technologies known for their versatility and low-power consumption, making them ideal choices for IoT applications.
- 802.16: Also known as WiMAX, 802.16 extends the reach of wireless networks, particularly in metropolitan and rural areas. It offers high data rates and robust connectivity, enhancing IoT capabilities.

Cellular Networks Cellular networks represent a pivotal facet of IoT connectivity, providing ubiquitous coverage across vast geographic areas. These networks consist of discrete *cells*, each served by a fixed-location transceiver known as a base station. This infrastructure enables the transmission of voice, data, and multimedia content. Cellular networks employ frequency division among cells to mitigate interference and ensure optimal service quality.

Wireless Connectivity Cellular networks employ a *cell* structure to cover extensive regions with radio waves. This architecture facilitates seamless communication for a plethora of portable transceivers, including mobile phones with mobile broadband modems. These devices traverse through multiple cells during communication, ensuring uninterrupted connectivity.

In the intricate tapestry of IoT connectivity, these generic protocols stand as robust pillars, offering diverse options for both local and wide-area communication. The IEEE 802 Standard, in particular, continues to evolve, providing a fertile ground for innovation and the seamless integration of IoT devices into modern networks.

LoRaWan In the realm of IoT connectivity, LoRaWAN stands as a beacon of innovation, offering a powerful and efficient solution through Low-Power Wide-Area Networking (LPWAN) technology. LoRaWAN has gained widespread adoption across the globe, finding application in diverse domains such as energy management, resource conservation, pollution control, infrastructure optimization, disaster mitigation, and more.

BlueTooth Bluetooth, a pioneering wireless technology, has revolutionized connectivity by facilitating the creation of Personal Area Networks (PANs). It accomplishes this by enabling the seamless transfer of data over short distances between a diverse array of devices, both stationary and mobile. This feat is achieved through the utilization of short-wavelength Ultra High-Frequency (UHF) radio waves, operating within the industrial radio bands. Bluetooth has emerged as a transformative force in wireless connectivity, reshaping the way devices interact within PANs. Its packet-based data transfer, frequency-hopping agility, and adaptability across a multitude of devices have made it an integral part of modern technology ecosystems. From personal devices to IoT applications, Bluetooth continues to empower seamless wireless communication.

Zigbee Zigbee stands as a formidable high-level protocol, purpose-built for the establishment of Personal Area Networks (PANs) that cater to specific applications such as home automation, medical device data collection, and scenarios with stringent requirements for low-power and low-bandwidth communication. In essence, Zigbee serves as an indispensable technology for applications demanding reliable, low-power, and secure wireless communication. Its ability to cater to a diverse range of use cases, cost-effectiveness, and emphasis on power efficiency make it an attractive choice in the realm of WPANs. Whether in home automation, healthcare, or industrial automation, Zigbee continues to enable connectivity solutions that thrive in resource-constrained environments.

LTE-M NgIoT The advent of Next-Generation Internet of Things (NGIoT) has spurred the development of cutting-edge technologies to underpin the ever-evolving landscape of connected devices. Among these innovations, LTE-M (Long-Term Evolution for Machines) has emerged as a pivotal low-power wide area network (LPWAN) tailored explicitly for machine-to-machine (M2M) and Internet of Things (IoT) applications. LTE-M NgIoT represents a pivotal advancement in IoT connectivity, aligning itself with the demands of an increasingly interconnected world. Its low-power, wide area coverage, enhanced data capabilities, mobility support, and compatibility with NGIoT technologies position it as a cornerstone for driving innovation in both public and private sectors. As the IoT landscape continues to evolve, LTE-M NgIoT stands as a testament to the relentless pursuit of more efficient, connected, and intelligent solutions.

Narrowband IoT (NB-IoT) The Third Generation Partnership Project (3GPP) has played a pivotal role in shaping the landscape of mobile telephony protocols[52]. Established to foster collaboration and standardization in the telecommunications industry, 3GPP's contributions have significantly expanded the capabilities of cellular services, paving the way for a new era of connectivity. 3GPP's concerted efforts in advancing mobile telephony protocols have not only expanded the realm of possibilities for cellular services but have also laid the foundation for the seamless integration of IoT devices into the cellular network. As the IoT ecosystem continues to flourish, the standards set by 3GPP remain instrumental in enabling efficient, low-power, and widespread connectivity for a myriad of IoT applications.

2.6.7. Storing and processing of data

A new era of data creation, storage, and processing has begun as a result of the proliferation of IoT devices. However, this abundance of data brings with it the challenge of effectively managing and securing it across diverse physical locations. In the realm of IoT, orchestrating data from myriad devices is a complex task, often requiring tailored strategies to maximize its value while ensuring data integrity and security.

Data Optimization Notably, IoT systems often generate vast amounts of data, but storing all this information in its entirety may not always be necessary. Consider a traffic camera that counts passing vehicles; what's crucial is the count data over specific time intervals, not the entire video feed. Herein lies the importance of data optimization, where irrelevant or redundant data can be filtered out or summarized before storage or transmission. This not only conserves storage space but also streamlines data processing.

Edge Computing IoT devices frequently operate in real-time environments and may require immediate decision-making capabilities. In such scenarios, it's impractical to transmit data to a central data center for processing, as this introduces latency. Enter edge computing, a paradigm that empowers IoT devices to process data locally, at the *edge* of the network. This approach enables rapid decision-making, reducing latency and dependence on central data centers. For instance, an autonomous vehicle may need to process sensor data instantly to navigate safely.

Distributed Data Processing With IoT, data is generated and processed across a distributed network of devices. This decentralization of data processing necessitates a shift in computational paradigms. Businesses must incorporate the capability to deploy computing and applications to the network's edge. This enables devices to pre-process data before transmitting it to central data centers for long-term analysis. For instance, a smart factory may locally analyze sensor data to optimize machinery performance before sending summarized data to the central monitoring system.

In the IoT landscape, the journey of data, from creation to processing and storage, is marked by complexity and dynamism. Effectively managing this data requires a holistic approach that considers optimization, edge computing, and distributed processing. As IoT continues to reshape industries and our daily lives, mastering the art of data handling is paramount for harnessing its full potential.

IoT Platform supporting connectivity

In the intricate ecosystem of the Internet of Things (IoT), platforms play a pivotal role as tools for creating, running, and managing applications. These platforms serve as the linchpin connecting the physical world of sensors and devices with the digital realm of data processing and application development.

Embedded Board Platforms Hardware manufacturers leverage embedded board platforms to craft IoT applications. These platforms offer a foundation for building intelligent solutions, often at the device level. Manufacturers use these boards to imbue devices with computational capabilities, enabling them to collect data and perform basic processing tasks.

Cloud-Centric Development For cloud service providers and application developers, IoT platforms provide a canvas upon which applications are woven using the data harvested from sensors and devices. These platforms often dwell in the cloud and offer a rich set of tools and services for handling IoT data. They serve as the intermediary layer between physical devices and actionable insights.

Data Consumption and Action At the core of IoT platforms is their ability to consume data from sensors and devices and transform it into meaningful actions. They provide well-defined Application Programming Interfaces (APIs) that empower developers to seamlessly connect diverse hardware platforms and harness cloud-based services.

Centralized Platforms IoT platforms can assume various architectures, with centralized platforms being one of them. In this model, a central hub, typically powered by the cloud, manages the operations of connected nodes or smart devices. While centralized platforms are effective for many IoT applications, they may fall short when dealing with the demands of industrial IoT solutions.

IoT Platform as a Service (PaaS) PaaS is a framework that empowers developers to expedite the creation and testing of IoT applications. It offers a structured environment for designing and building applications efficiently. PaaS streamlines the development process by managing underlying components like operating systems and virtualization. It significantly reduces coding efforts and automates compliance with company policies, providing scalability, high availability, and multi-tenancy in a cloud-based context.

Decentralized Platforms In contrast, decentralized platforms are ushering in the next wave of IoT innovation. These platforms enable nodes within an IoT network to interact autonomously, free from the constraints of centralized authority. Decentralization brings several advantages to IoT, including robustness, scalability, low power consumption, streamlined data and device management, and the integration of artificial intelligence at the network's edge.

Five Pillars of Decentralized IoT Decentralized IoT architectures are underpinned by five key pillars: a multi-network approach, scalable and interoperable implementation, efficient power management, intuitive data and device administration, and the infusion of artificial intelligence at the edge. These attributes collectively empower IoT platforms to tackle a broader spectrum of applications and enable more efficient, resilient, and intelligent IoT ecosystems.

In the evolving landscape of IoT, platforms remain the catalyst for innovation, connecting disparate elements into cohesive and intelligent systems that drive our interconnected future.

IoT data storages

The deluge of data generated by the Internet of Things (IoT) poses a complex challenge: how to efficiently collect, store, and manage this wealth of information. Open-source platforms, such as ThingSpeak, have emerged to address this issue, offering tools for sensor data collection and cloud integration. Some platforms, like ThingSpeak, even provide dedicated Matlab apps for data analysis and visualization. Popular IoT boards like Beaglebone, Raspberry Pi, and Arduino facilitate the transmission of sensor data, which can be directed to specific channels for organization and analysis.

The IoT landscape boasts a variety of platforms, each tailored to unique needs. Some notable contenders include Thingworx 8 IoT Platform, Microsoft Azure IoT Suite, Google Cloud IoT Platform, IBM Watson IoT Platform, AWS IoT Platform, Cisco IoT Cloud Connect, Salesforce IoT Cloud, Kaa IoT Platform, and Oracle IoT Platform. These platforms offer a spectrum of features, enabling businesses to select the one that aligns with their IoT objectives.

Edge Computing as the IoT Data Solution: IoT data management and archival present a formidable challenge, particularly when dealing with remote locations, like branch offices or plant machinery that operated through servers. In such cases, edge computing emerges as the go-to solution. Edge computing entails processing and managing operations outside the centralized data center.

The rising popularity of edge computing is attributed to its capability to handle the vast volumes of data generated beyond centralized data centers. To ensure secure IoT data processing and storage, several considerations come into play. Firstly, substantial investments are required in external networking infrastructure. Secondly, data aggregation may suffice, obviating the need to store all device data. Lastly, real-time data processing is imperative.

Keeping all IoT data within data centers undermines the feasibility of these prerequisites. This is where Information Lifecycle Management (ILM) steps in. ILM extends beyond cost-effective data storage, ensuring data resides in locations conducive to immediate processing, leveraging machine learning and AI algorithms for valuable business insights.

Current Options for IoT Data Processing and Storage: The IT industry presents a myriad of solutions for processing and storing IoT data effectively:

- **Public Cloud Storage**: Public cloud providers not only offer storage but also empower users to process and analyze data using AI tools.
- **Snowball Appliance**: IoT supplier solutions like Amazon Web Services (AWS) employ the Snowball appliance, a server with storage, to physically transfer and locally process data from offsite locations.
- **Google Cloud Platform (GCP)**: GCP offers a cloud processing platform for unstructured IoT data.
- **On-Premise ML/AI Systems**: Companies like DDN have engineered converged infrastructure solutions to store and analyze IoT data using on-premise ML/AI systems.

- **Excelero’s Scalable File Storage**: Excelero has developed a product to meet the demands of scalable file storage and low-latency analytics in the IoT landscape.

The IoT data storage and processing realm is a dynamic space, constantly evolving to meet the demands of the IoT revolution. As the IoT continues to expand, the need for efficient, secure, and scalable data solutions becomes increasingly vital.

2.6.8. Connected appliance design from electronics to firmware

The rapid expansion of the Internet of Things (IoT) and its associated services is poised to usher in a new era of ubiquitous computing and communication. Over the past few years, devices have undergone a significant transformation. They have shrunk in size while experiencing a substantial boost in computational power, causing them to seemingly ”disappear” into the fabric of our lives, seamlessly integrating into larger systems that can control our environments, such as networks of home appliances.

Embedded electronics are at the heart of these connected home appliances. These electronics comprise devices equipped with CPUs, actuators, and sensors to ensure real-time functionality. Given their safety-critical nature and the potential for catastrophic consequences in terms of human life and the environment, these appliances impose stringent requirements for reliability and assured performance.

However, the majority of household appliances currently lack wireless communication interfaces and microcontrollers, rendering them incapable of connecting to the Internet. Recent advancements in embedded computing are changing this landscape. As a result, the IoT is on the cusp of witnessing a surge in interconnected white goods, as household appliances increasingly adopt wireless connectivity. This shift promises to revolutionize our daily lives by enhancing the efficiency and convenience of these appliances through seamless integration into the IoT ecosystem.

Requirements for Efficient and Reliable IoT Appliance Design

When it comes to crafting efficient and dependable IoT home appliances, or any connected device, two primary components must be considered: electronics and the communication module. These elements are central to ensuring efficient operation and the perception of reliable radiofrequency signal strength. The design of an IoT appliance involves the seamless integration of software development across various levels (refer to Figure 2.22). This often highlights significant challenges faced by development teams, especially in organizations transitioning from traditional development paradigms to service-oriented ones.

At its core, any connected or even standard non-connected device’s hardware includes a user interface for appliance control and monitoring. This interface is linked to the power board, which frequently manages auxiliary components such as heating and actuation devices. While some appliances may involve

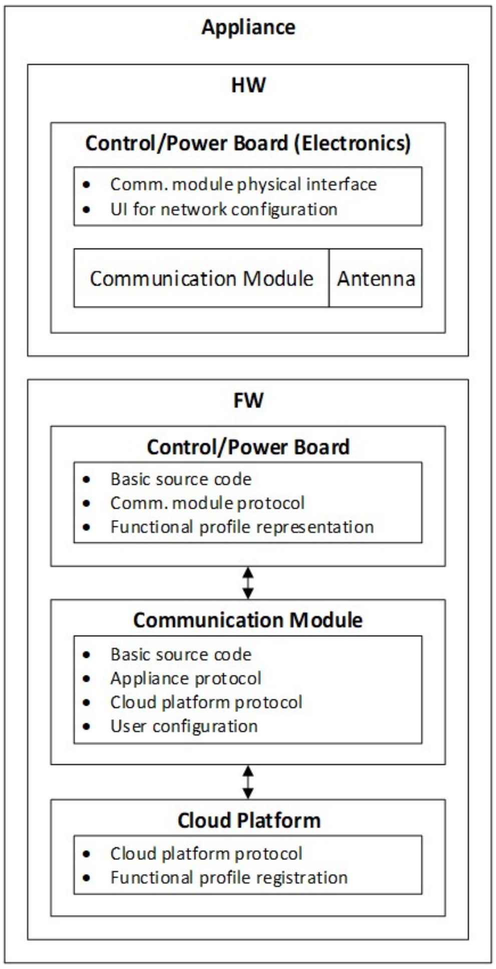


Figure 2.22.: *The elements that compose the connected appliance*

multiple electronics cooperating to oversee their operation, the control board stands out as the most common electronic component housing the communication module. The essential requirements for hosting electronics encompass the physical interface connecting the communication module (e.g. UART) and the user interface, which may feature additional functionality for configuring the module's network parameters.

In contrast, the development of software for connected appliances demands a profound understanding of the entire sequence of actions, from the appliance itself to the end-user application. Software maintenance assumes paramount importance in the realm of connected appliances and can result in significant unforeseen costs if not managed efficiently. Well-designed hardware, if properly engineered, should not necessitate maintenance and should avoid incurring additional expenses.

Furthermore, the development of connected appliances entails creating a functional profile representation, essentially a dictionary of functionalities. It also involves implementing the communication protocol that facilitates connectivity between the host electronics and the communication module. The communication module requires two protocol implementations: the first links the module to the host electronics, while the second connects it to a cloud platform that registers appliances based on their functional profiles. This comprehensive approach ensures the efficient and reliable design of IoT appliances, paving the way for a seamless user experience.

Definition of functional profile

In the realm of requirements engineering for Internet of Things (IoT) infrastructures, the definition of a functional profile assumes paramount importance. Essentially, a functional profile serves as a meticulously structured representation, often implemented using mark-up languages or similar formats, that comprehensively elucidates the capabilities and features of an embedded device. This profile functions as an exhaustive dictionary of functionalities, encompassing various attributes that meticulously characterize the distinct functionalities of a given device. As delineated by Saso [53], these attributes typically encompass:

1. **Name of the Functionality**: This denotes the nomenclature or reference used to identify a specific function within the device.

2. **Functionality Identifier**: A unique identifier assigned to each functionality to ensure unambiguous reference.

3. **Type of Value**: Specifies the nature of data or information associated with the functionality, whether numerical, textual, boolean, etc.

4. **Type of Access to the Value**: Specifies how the data or value linked with the functionality can be accessed, whether it's read-only, write-only, or both.

5. **Automatic Synchronization**: Indicates whether the functionality supports automatic synchronization of its data with other devices or systems.

6. **Initial Value**: Specifies the default or initial value for the functionality when the device is activated or initialized.

7. **Range of Valid Value**: Defines the acceptable or allowable range within which the values connected with the functionality must reside.

Table 2.4 provides an illustrative excerpt of this functional profile, particularly within the context of a specific category of household appliances, such as cooling devices. This table essentially presents a structured representation of functionalities, their attributes, and potentially sample data pertaining to a subset of appliances belonging to this category. It serves as a practical example of how a functional profile is instrumental in IoT requirements engineering, facilitating the documentation and comprehension of device capabilities.

Functionality	Identifier	Access	Type	Valid Values
DEVICE_STATUS	2010	Read-only	Enumeration	2011 = IDLE 2012 = SERVICE
TEMPERATURE	2070	Read-only	Signed Integer	The value is read from the NTC sensor
DOOR_STATUS	2090	Read-only	Enumeration	2091 = CLOSED 2092 = OPEN

LIGHT_STATUS	2130	Read-write	Enumeration	OFF = 2131
				ON = 2132

ALARM	2160	Read-only	Enumeration	2161 = NONE 2162 = DOOR_OPEN

COMPRESSOR	2330	Read-write	Enumeration	OFF = 2331
				ON = 2332

Table 2.4.: *Refrigerator functional profile*

Hardware Design for IoT-Enabled Household Appliances: Striking a Balance In the domain of hardware design for IoT-enabled household appliances, an intricate balance between the ultimate product cost and the viability of implementing a standardized solution across diverse appliance categories and models stands as a contemporary challenge of significant proportions. This challenge encompasses the hardware or electronic components of the appliance and mandates adherence to specific requirements, which encompass:

1. **Electronics' Interface for Communication Module**: This domain encompasses the intricate design and seamless integration of the appliance's electronic interface, ensuring its harmonious compatibility and connectivity with the communication module. This interface stands as the pivotal conduit for the exchange of data and communication between the appliance and external systems or networks.

2. **Optimal Placement of Communication Module's Antenna**: The strategic determination of the most advantageous location for the communication module's antenna within the appliance looms as a critical consideration. This placement necessitates a meticulous evaluation of factors such as signal strength, interference mitigation, and the overarching optimization of performance. These factors converge to guarantee the steadfast reliability of communication.

3. **User Interface Augmentation for Communication Module Configuration**: To facilitate user-friendly setup and configuration of the communication module, a supplementary user interface component must be thoughtfully woven into the appliance's design fabric. This augmentation empowers users with the convenient means to tailor and fine-tune communication settings in accordance with their preferences and specific network requisites.

Conscientiously addressing these hardware design requisites not only ensures the seamless integration of IoT functionalities into household appliances but also streamlines production costs and lends support to a more unified approach that can be universally applied across an array of appliance categories and models. This alignment with principles of standardization assumes a pivotal role in fostering interoperability and scalability within the dynamic IoT ecosystem.

Firmware design In the domain of firmware design for IoT-equipped household appliances, it is imperative to adhere to practices that prevent any disruption to the core source code of the appliance. The connected code should act as an encapsulating layer or wrapper around the existing source code. Within this context, there exist three fundamental pillars that warrant careful consideration:

1. Protocol for Electronics-Communication Module Connectivity: The selection and implementation of a robust communication protocol are paramount. This protocol serves as the conduit for seamless interaction between the appliance's electronic components and the communication module. Its efficiency and reliability are pivotal in ensuring data exchange and synchronization.

2. Functional Profile and Internal Protocol Implementation: To enable the comprehensive gathering of information from multiple electronic components and sensors within the appliance, it is essential to define a well-structured functional profile. This profile delineates the various functionalities, their identifiers, data types, access mechanisms, automatic synchronization procedures, initial values, and valid value ranges. The internal protocol implementation should adhere to these specifications, facilitating efficient data collection and transmission.

3. Firmware Update Mechanism: Implementing a firmware update mechanism is crucial for maintaining the appliance's functionality and security over time. This mechanism can be achieved through either wired or over-the-air (OTA) solutions. However, due to the diverse nature of electronic components and configurations across various appliances, devising a firmware update strategy can present challenges. Ensuring compatibility and reliability across this diversity is a primary concern.

By meticulously addressing these three pillars in firmware design, household appliances can seamlessly embrace IoT capabilities without compromising their core functionality. This approach fosters enhanced connectivity, data acquisition, and the ability to adapt to evolving requirements, all while preserving the integrity of the appliance's essential operations.

Communication module

Security aspect The security aspect in the context of communication between the appliance's electronics and the communication module is of utmost importance, primarily due to safety concerns and the potential for security breaches. To ensure the integrity and confidentiality of the data, it is highly recommended that the data is properly encrypted during transmission. However, there are certain challenges to consider. To keep costs down for the final product, manufacturers often opt for tailored microcontrollers and limited memory capacity. Implementing a custom encryption protocol under these constraints can be a significant challenge, despite its critical importance.

Operating temperature and dimensions In household appliance development, operating temperature is typically not a significant concern. However, in specific categories like cooking appliances (e.g. ovens and hobs), temperature becomes a critical parameter. The prevailing trend is to design modules with the smallest possible dimensions. This approach is driven by the need for cross-category unification. Nevertheless, it's essential to strike a balance, as larger dimensions could impact mounting in certain cases.

Furthermore, for the sake of standardization and unification across different appliance categories, it's advisable that the communication module is certified to operate reliably at elevated temperatures, typically at least 85°C. This certification ensures that the module can function effectively across a range of household appliances, including those that generate significant heat during operation, without compromising safety or performance.

Testing and certification Comprehensive testing and certification are crucial steps in ensuring the reliability and market success of a communication module for household appliances. Rigorous testing should encompass various environmental factors such as exposure to high temperatures, humidity, and vibration. Detecting and rectifying potential issues during testing is essential, as any failures in the field could have severe consequences for the product's reputation and market performance.

Certification plays an equally significant role and goes hand-in-hand with testing. Accumulating a diverse range of certifications demonstrates the module's compliance with industry standards and safety regulations. In the competitive landscape of household appliances, having an extensive list of certificates can significantly enhance the module's market prospects. It instills confidence in both manufacturers and consumers, assuring them of the module's reliability and adherence to stringent quality standards.

Firmware development The development of firmware for the communication module in household appliances is a critical aspect, serving as the vital link between the physical appliance and the external world. Designing the firmware with a focus on extensibility and ease of maintenance is paramount for long-term success. Several key components should be carefully considered during firmware development:

- Protocol for Electronics and Communication Module: The firmware should include a robust protocol for seamless communication between the appliance's electronics and the communication module. This protocol ensures efficient data transfer and synchronization.

- Protocol for Communication Module and Cloud Platform: Another crucial component is the protocol that facilitates communication between the communication module and the designated cloud platform. This connection enables data transmission, storage, and retrieval from remote servers, enhancing the appliance's functionality.
- User-Friendly Configuration: To enhance user experience, the firmware should provide a user-friendly configuration interface. This could be accessible through a website or a dedicated application, allowing users to easily set up and customize the module's functionality.

By carefully addressing these firmware development aspects, manufacturers can ensure that their communication modules are adaptable, maintainable, and user-friendly, thereby meeting the demands of both consumers and the rapidly evolving Internet of Things (IoT) landscape.

Power consumption and power management Power consumption is a critical consideration in the development of household appliance communication modules, especially in light of evolving legislative and regulatory demands that emphasize energy efficiency. Manufacturers are faced with the challenge of balancing the power requirements of the module with increasing customer demands for connectivity features. These demands can significantly impact power consumption and create concerns for manufacturers.

One effective strategy to address this issue is the implementation of a robust power management protocol. The key recommendation is to define and integrate this protocol at the outset of the project. The primary goal of such a protocol is to mitigate the rise in power consumption associated with enhanced connectivity features.

The power management protocol can achieve this by incorporating the following strategies:

1. Scheduled Module Power Cycling: The protocol can schedule times when the communication module is turned off. During these periods, the module remains in a low-power state. It can then be periodically awakened to receive messages sent by the cloud platform or a dedicated application.
2. User-Initiated Power Activation: The protocol should also account for user interaction. For instance, when a user interacts with the appliance by pressing control buttons or making adjustments through the control panel, the communication module can be automatically activated to respond to these actions.

By implementing a well-designed power management protocol, manufacturers can strike a balance between delivering the desired connectivity functionalities and minimizing power consumption. This approach aligns with both energy efficiency regulations and consumer expectations for modern, eco-friendly appliances.

Mounting and antenna positioning A modular approach, featuring a single communication module, holds promise for maintaining cost-efficiency and standardizing procurement and stock-keeping practices. When considering the integration of the module with the appliance's host electronics, three main options are available:

1. Direct Connection via Connector: This option offers portability between different appliances, but it limits the antenna's position to that of the host electronics.
2. Direct Soldering: Direct soldering is a viable choice for applications with higher demands and substantial manufacturing volumes. However, it can pose challenges for after-sales replacement in the event of malfunction.
3. Indirect Mounting with a Connecting Cable: In appliances with limited mounting space or specific design constraints, an indirect approach using a mounting case and connecting cable may be considered. This option can offer flexibility in installation.

Optimizing the positioning of the module's antenna is crucial for achieving the best possible signal strength. Identifying the optimal antenna location can be challenging due to the diverse design constraints of different appliances, such as standalone versus built-in refrigerators or ovens with glass versus metallic control panels.

While external antennas are a possibility, they introduce additional costs related to cabling and connectors. Therefore, a preferred choice is to design the module with an on-board antenna to minimize costs and streamline integration. Proper antenna placement is a critical aspect of ensuring reliable communication for connected appliances.

2.6.9. Preliminary Conclusions

In this chapter, we have delved into the essential components of IoT infrastructure, including IoT gateways, middleware, and cloud platforms. Our exploration began with an overview of selected platforms, highlighting their distinctions to assist in platform selection for developing IoT devices. It's important to emphasize that each IoT project is unique, and the choice of platform should align with the specific requirements of the project.

Furthermore, we delved into the realm of communication protocols, which serve as the backbone for managing data within IoT infrastructures. These protocols play a crucial role in ensuring seamless data exchange between devices. We also examined the intricacies of data storage and processing within the IoT ecosystem, emphasizing the significance of efficient data management.

The final section of this chapter centered on the design of connected appliances, spanning from hardware design considerations to firmware development. This holistic approach underscores the importance of meticulous planning and execution to create IoT devices that meet both functional and efficiency requirements.

As we move forward, the next chapters will build upon this foundational knowledge, exploring advanced topics and practical applications within the vast and dynamic field of the Internet of Things.

2.7. The VICINITY project

The VICINITY project stands as a pioneering initiative within the Horizon 2020 program, executed between 2016 and 2020, with a core objective of establishing an innovative platform aimed at connecting and unifying disparate IoT infrastructures. This unified ecosystem, often referred to as a *virtual neighborhood*, empowers users to make selections concerning the interconnection of their smart objects within a peer-to-peer network. It effectively transforms isolated IoT infrastructures into interoperable systems, operating seamlessly at both technical and semantic levels. A pivotal development underpinning this transformation is the seamless integration of a sophisticated semantic model into the VICINITY platform, greatly enhancing semantic interoperability among smart objects originating from various operators and adhering to diverse standards [54].

The architectural foundation of the VICINITY platform, which advocates for interoperability as a service, is elegantly depicted in Figure 2.23. This architectural framework embodies a decentralized, bottom-up, and cross-domain approach, reminiscent of a social network. Users are empowered to configure their IoT setups, seamlessly integrating standards aligned with their chosen services. They have full control over their privacy levels within the peer-to-peer network. In essence, VICINITY empowers users to orchestrate their IoT ecosystems to meet their unique needs and preferences. By amalgamating services from diverse domains and granting users the authority to define the extent of information sharing while respecting privacy concerns, the VICINITY platform fosters synergies among services from various domains. This visionary approach holds the potential to usher in an era of novel cross-domain services, thereby expanding the horizons of the IoT market.

Upon the completion of the VICINITY project, the platform's efficacy and innovation were comprehensively demonstrated through pilot sites across diverse domains, including energy, building automation, health, and transport. The project showcased the platform's ability to create cross-domain services that generate added value for users. Notable examples of these value-added services include micro-trading of demand-side management capabilities, AI-driven optimization of smart urban districts, and the application of business intelligence principles to IoT. These achievements underscored VICINITY's potential not only to harmonize the intricacies of IoT but also to unlock the substantial potential of cross-domain services.

This dissertation focuses primarily on the creation of a virtual environment designed for testing and validating IoT infrastructures and the use cases that go along with them. This virtual environment accurately replicates real-life scenarios before their physical deployment. Within this context, a prominent use case pertains to smart energy management at one of VICINITY's pilot

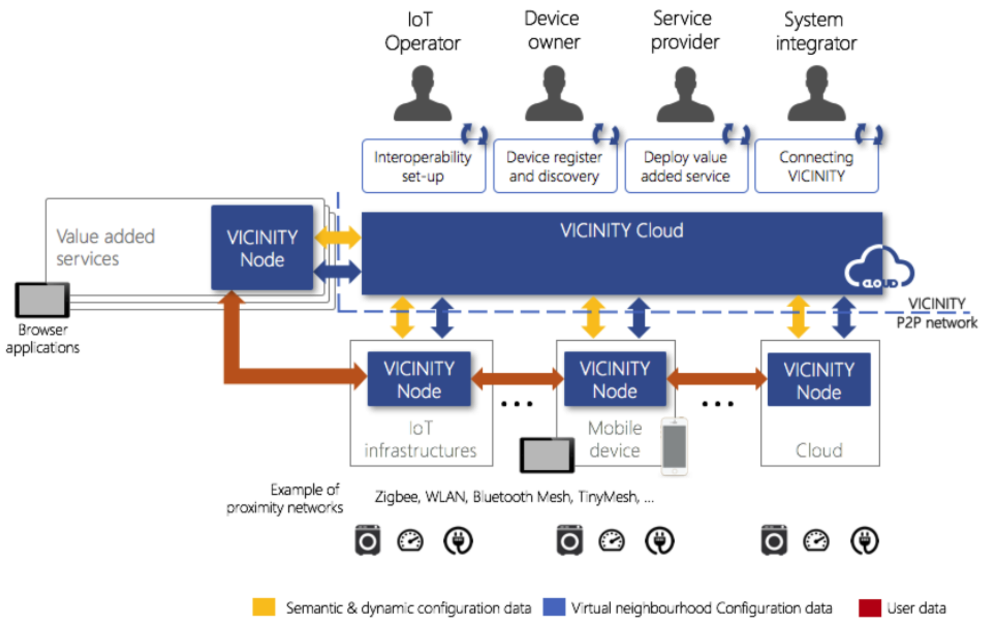


Figure 2.23.: *High-level VICINITY architecture [55]*

sites in Tromsø, Norway. This smart energy use case serves as an illustrative example of VICINITY's potential to foster innovative and efficient IoT solutions.

2.8. Homomorphic encryption

Within the architecture of VICINITY, the paramount importance of privacy protection is enshrined, ingeniously designed to ensure that only metadata pertaining to connected devices is stored within a central cloud. Sensitive and personal data, exemplified by sensor readings, is transmitted exclusively on a peer-to-peer basis, traversing directly from the data producer to its intended consumer. The transmission of such sensitive data requires prior and individual approval from the data owner, affirming their explicit consent.

However, even within this meticulously designed privacy framework, challenges persist. Once an individual has granted consent for data sharing, their information becomes accessible to potential third-party entities, some of which may initially appear trustworthy but could harbor malicious intent. In the context of IoT, the concept of "Value-Added Services (VAS)" stands as a linchpin, essential for ushering the full potential of smart IoT applications. Users, despite harboring reservations about divulging their personal information, may find it tempting to share their data with VAS providers, ultimately compromising their privacy.

A far more desirable approach, one that obviates the need to expose sensitive data in plain text while still facilitating the functionality of VAS, can be realized through the application of homomorphic encryption (HE) schemes.

Homomorphic encryption presents a remarkable capability: the execution of specific calculations (or in the case of fully homomorphic encryption, arbitrary functions) on encrypted ciphertexts, all without the necessity of decrypting the data beforehand. This cryptographic technique, once considered an enigma, is the linchpin for preserving data privacy.

Historically, partially homomorphic encryption schemes were the focus of research efforts. These schemes, while valuable, had limitations, constraining the types of operations that could be executed on encrypted data. However, a monumental breakthrough occurred with the introduction of fully homomorphic encryption by Gentry in 2009 [56, 57]. This innovation empowered the execution of arbitrary computations on ciphertexts without any requirement to decrypt the data, thereby eliminating the exposure of plaintext information.

It is essential to recognize that the adoption of fully homomorphic encryption does entail computational costs. The increased computational demand necessitates a trade-off between performance and versatility, and organizations must make informed decisions in this regard.

In Chapter 5.2.2, we will delve into practical applications of homomorphic encryption within our framework. This exploration will allow us to assess and evaluate various options within a realistic yet controlled environment, thereby shedding light on the intricate interplay between user privacy, data utility, and computational resources in the context of VICINITY's IoT ecosystem.

Chapter 3

State of the Art

Contents

In the realm of IoT and Cyber-Physical Systems, this chapter embarks on a comprehensive exploration of the state-of-the-art in simulation techniques. It commences with a deep dive into the simulation of the Internet of Things (IoT), unearthing the key requirements that underpin effective IoT simulation. This journey further unravels the intricacies of crafting simulation scenarios in the context of IoT, while simultaneously addressing the multifaceted challenges and shedding light on future directions in this dynamic field. The chapter then extends its focus to the simulation of Cyber-Physical Systems (CPS), emphasizing the critical aspects of validation, verification, and the pivotal role of Discrete Event Simulation in CPS. A panoramic view of existing IoT simulators is presented, from stalwarts like S^3 and OMNeT++ to innovative hybrids like ACOSO and OMNeT++. The chapter's voyage continues with a spotlight on MAMMotH for emulating large-scale IoT scenarios and explores the nuances of DEUS, COOJA, and NS3. The chapter culminates in a comparative analysis of IoT simulation approaches, dissecting interconnected simulation levels, versatility, community support, synchronization, scalability, and continuous-time approximations. This comprehensive overview sets the stage for an in-depth understanding of the evolving landscape of IoT and CPS simulation methodologies.

3.1. Simulation of the Internet of Things

The development and validation of Internet of Things (IoT) systems rely heavily on simulation, particularly given how diverse and dynamic IoT networks are. These simulations are essential tools for testing and optimizing different IoT system components, such as communication protocols, energy consumption patterns, and decision-making processes in a variety of scenarios. The basic requirements, difficulties, and developments in this field are highlighted in this chapter as we dig into the multifaceted world of Internet of Things simulation. In order to establish the platform for a detailed examination of its intricacies, this part gives a fundamental grasp of IoT simulation.

The Internet of Things (IoT) has revolutionized how devices interact and share information, ushering in an era of interconnected smart systems. However, this intricate web of devices, each operating with distinct standards and functionalities, poses significant challenges when it comes to testing and validating IoT ecosystems. IoT simulation emerges as a powerful tool to address these challenges, offering a controlled environment to evaluate and optimize IoT solutions.

Key Requirements for IoT Simulation A. Gluhak, in the paper titled "A Survey on Facilities for Experimental IoT Research" [58], outlines crucial requirements for effective IoT simulation:

1. **Scalability:** The exponential growth of IoT devices necessitates simulations capable of handling thousands of nodes, accommodating the rapidly expanding IoT landscape.

2. **Heterogeneity:** IoT environments encompass highly diverse devices, and a robust simulator must support this heterogeneity while ensuring that device programmability remains manageable.

3. **Repeatability:** Simulations should be replicable across different testbeds, promoting consistency and facilitating cross-testbed comparisons. Achieving this objective mandates standardization and the consistent packaging of simulation results.

4. **Federation:** Complex scenarios often necessitate distributed simulations. A common framework for authentication and experiment scheduling is vital to connect and coordinate different simulations.

5. **Concurrency:** Many IoT applications involve multiple users interacting concurrently. Effective IoT simulations must minimize interference among concurrent experiments and efficiently allocate testbed resources.

6. **Mobility:** IoT devices frequently operate in dynamic, real-world environments. Simulators must emulate real-world device movements and interactions effectively.

7. **User Involvement and Impact:** IoT applications often require human interaction, a challenging aspect to simulate. To address this, simulations must support real-time execution to incorporate genuine user interactions seamlessly.

IoT simulation stands as a critical enabler in the development and validation of IoT solutions. By embracing the core requirements outlined by A. Gluhak and adapting to the evolving IoT landscape, simulations continue to play an instrumental role in fostering innovation, reliability, and security within the IoT ecosystem. This chapter sets the stage for a deeper exploration of IoT simulations, shedding light on their significance and potential in shaping the IoT landscape of tomorrow.

3.2. Simulation of Cyber-Physical Systems

The simulation of Cyber-Physical Systems (CPS) serves as a potent tool for the development and testing of various scenarios within the realm of the Internet of Things (IoT). Broadly, a simulation comprises two fundamental components: a model and the execution of a range of experiments. While simulations can theoretically be conducted with or without computer assistance, contemporary usage predominantly revolves around computerized simulations. Within this context, diverse programming and descriptive languages, spanning from low-level to system-level modeling languages, cater to specific use cases. This classification hinges on three primary model types, elucidating the underlying concept [59]:

- Physical models pertain to tangible entities, including objects such as a race car's gearwheel, a building's architectural layout, or even more abstract constructs.

- Decision-making models become instrumental in simulations where events or decisions singularly dictate the course of the process. This category finds common ground in sandtable exercises and coffee automata simulations.

- Deterministic models encompass stochastic models grounded in probability theory. They find application in areas like quantum theory or thermal flow simulations.

Figure 3.1 offers an overview of the relationships inherent in a simulation. The source system constitutes the real-world subject under investigation, situated within the experimental framework. This framework is supplemented by a behavior database sourced from prior observations or experiments. Out of this experimental context, a model is crafted to encapsulate a simplified version of reality. Subsequently, the simulator is forged from the model to generate the model's behavior. Given these intricate relationships, a closer examination of validation and verification becomes imperative.

3.2.1. Validation and Verification

Validation concerns itself with the modeling relation and gauges the degree of accuracy in representing the source system through the model. Conversely, verification pertains to the simulation relation and assesses the fidelity between the model and the simulator [60]. In essence, simulations are underpinned by mathematical models and serve as invaluable tools for predicting technical or economic processes, including failures. Historically, time-dependent models described using differential equations were the go-to choice for simulations [60]. However, the ever-increasing computational power has ushered in the era of complex simulations, particularly decision-making models.

3.2.2. The role of Discrete Event Simulation in CPS

In the context of CPS, where the dimension of time, particularly in terms of durability, is a critical parameter, discrete event simulations have emerged as the method of choice for drawing meaningful conclusions from simulations. These simulations are based on mathematical models and serve as a reliable means for predicting the behavior of systems, especially those within the IoT domain, where events play a pivotal role.

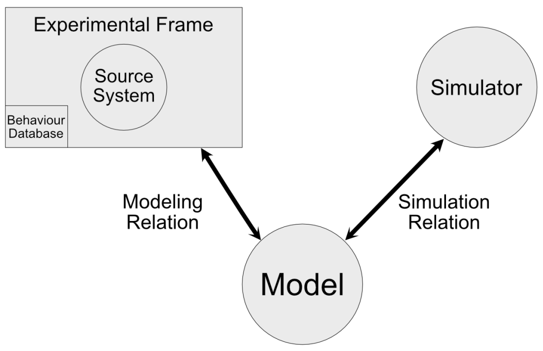


Figure 3.1.: *Relationships between the source system, model and simulation[60]*

In conclusion, the simulation of Cyber-Physical Systems is an indispensable tool within the IoT landscape. It encompasses diverse modeling approaches and techniques to capture the intricacies of real-world scenarios. These simulations are underpinned by mathematical models, enabling predictions, and assessments of technical and economic processes. With the rise of computational power, simulations have evolved to tackle complex decision-making models. In the ever-expanding IoT ecosystem, discrete event simulations have taken center stage due to their ability to faithfully replicate real-world dynamics, ensuring accurate verifications of CPS. This chapter serves as a foundational exploration of the significance and methodologies employed in simulating Cyber-Physical Systems, setting the stage for deeper investigations into this critical facet of IoT research and development.

3.3. Overview of IoT Simulators

The rapid and exponential growth of the Internet of Things (IoT) market underscores the need for robust tools and methodologies to address the increasing complexity of IoT systems. Presently, the number of interconnected devices has surpassed the global human population, illustrating the sheer scale and magnitude of this technological phenomenon [61]. In the intricate landscape of IoT networks, simulation emerges as a pivotal instrument, facilitating crucial functions such as early-phase validation before the deployment of IoT systems in real-world scenarios.

The IoT market is marked by its continuous expansion, with no signs of abating. The proliferation of connected devices, each contributing to the ever-expanding ecosystem, necessitates a comprehensive understanding of their behavior, interactions, and performance. As a result, simulation frameworks and platforms become indispensable in addressing the multifaceted challenges inherent to IoT networks.

The intricate nature of IoT networks, characterized by diverse devices, protocols, and dynamic environments, renders real-world testing and validation a costly and time-consuming endeavor. Simulation, on the other hand, offers a controlled and efficient means of replicating IoT scenarios, enabling researchers and practitioners to assess system functionality, analyze performance, and validate various use cases without the need for physical deployment.

Simulation serves as a critical tool in the early phases of IoT system development, where the feasibility and effectiveness of proposed solutions can be rigorously evaluated and refined. By emulating real-world conditions and scenarios, simulation allows for the identification of potential bottlenecks, vulnerabilities, and optimization opportunities, thereby contributing to the enhancement of IoT systems' robustness and reliability.

Furthermore, IoT simulation platforms foster innovation and drive research initiatives within the IoT domain. They provide a flexible and scalable environment for exploring new protocols, algorithms, and technologies. Researchers can experiment with various configurations, test hypotheses, and develop novel solutions, all within the confines of a simulated IoT ecosystem.

This section provides an insightful overview of IoT simulators, delving into the diverse range of tools and frameworks available for simulating IoT environments. It highlights their significance in IoT development and deployment and explores their contributions to research, testing, and validation within the IoT domain. Additionally, it offers valuable insights into the evolving landscape of IoT simulation, setting the stage for a comprehensive exploration of existing simulation platforms and their applications.

3.3.1. S^3 and OMNeT++

The concept of "Smart Shires" introduced by Feretti and D'Angelo [62] signifies a novel approach to harnessing the potential of the Internet of Things (IoT) in the development of intelligent, interconnected regions. Central to this approach is the deployment of simulation as a pivotal tool for architectural validation before the deployment of physical prototypes. Feretti and D'Angelo underscore the importance of scalability and real-time capabilities in their choice of simulation tools. Furthermore, they advocate for multi-level simulations, acknowledging that simulating the entire model at the highest level of detail is often impractical. Instead, their approach amalgamates distinct simulators, each specializing in simulating a specific domain.

This concept laid the foundation for the development of the "Smart Shire Simulator" (S^3), implemented using the "GAIA/ART'IS" middleware. The ART'IS framework is designed for executing large-scale simulations sequentially, in parallel, and in a distributed manner. It encompasses various communication algorithms, including TCP/IP, MPI, and shared memory, and offers synchronization mechanisms for both pessimistic and optimistic approaches. In contrast, the GAIA framework simplifies the simulation of scenarios in parallel and distributed fashion by providing a high-level application programming interface (API) that reduces simulation time through adaptive partitioning of the model.

Performance evaluation of S^3 confirmed its scalability limitations in a sequential setup, prompting the exploration of parallel configurations. Tests revealed that for moderate loads (ranging from 1000 to 8000 simulated entities), employing only two processor cores did not yield substantial speedup due to communication overhead. In cases of heavy loads, additional cores demonstrated improved performance, albeit not commensurate with expectations. This subpar performance is attributed to the nature of the model, characterized by limited computational requirements per simulated entity but a high volume of inter-entity communications. Subsequent experiments evaluated the impact of adaptive partitioning on performance. Results consistently indicated speedup relative to static partitioning, reinforcing its utility in optimizing simulation efficiency.

In their subsequent work [63], D'Angelo, Feretti, and Ghini extended their simulation framework to encompass the simulation of the Internet of Things (IoT), merging Smart Shires with surrounding urban areas. Here, the authors proposed an intricate multi-level simulation approach, where a high-level simulator orchestrates domain-specific simulators. They emphasized the importance of inter-model interactions and interoperability between simulators in this complex setup. The authors applied this approach to a "Smart Market" scenario, introducing a more advanced version of multi-level simulation that included sequential execution of different simulators.

The top-level simulation was implemented using S^3 with improved broadcasting via "geoPbB". Level 1 incorporated two simulators in succession: an ADVISOR-based simulator, built on MATLAB/Simulink for analyzing vehicle performance and fuel economy, and an OMNeT++ simulator for modeling grid-based market sellers and mobile pedestrian nodes. The top-level and lower-level simulators communicated through TCP socket connections. Experiments confirmed that more logical processes in level 0 increased memory consumption, highlighting the necessity of partitioning level 1 across multiple interconnected hosts.

Building on this multi-level simulation approach, D'Angelo, Feretti, and Ghini further explored the potential of Parallel and Distributed Simulation (PADS) in their work [64]. They recognized the scalability limitations of conventional monolithic simulators and proposed a hybrid simulation approach. This approach aimed to mitigate the challenges associated with scaling the number of simulated entities in large-scale IoT simulations. While advocating the use of hybrid simulators, the authors highlighted potential concerns related to inter-model interactions, data transfer between simulators, and introduced approximation errors.

To evaluate the efficiency of their proposed hybrid approach, the authors conducted experiments with S^3 and OMNeT++ individually. Their findings supported the assumption that increasing logical processes in level 0 resulted in a substantial rise in memory consumption. They suggested partitioning level 1 across multiple hosts as a solution. Subsequent experiments scrutinized the parallel and distributed approach, with the top-level (S^3) and level 1b (OMNeT++) residing on a Linux host, and level 1a (ADVISOR[65]) on a Windows host. Experiments revealed that introducing multiple logical processes in level 0 incurred significant overhead. The authors concluded that while this approach demonstrated promise for scalability, it was imperative to carefully orchestrate the synchronization of logical processes to avoid memory thrashing.

This work extends the concept of multi-level simulation to exclusively employ discrete event simulation techniques while fostering interoperability between diverse simulators.

Multi-Level Simulation for Scalable IoT Modeling

The multi-level simulation approach, as presented in previous research by Feretti and D'Angelo [62] and further expanded upon by D'Angelo, Feretti, and Ghini [63] and [64], serves as a powerful paradigm for scalable modeling of the Internet of Things (IoT). This section delves deeper into this approach, elucidating its intricacies and highlighting its advantages for simulating complex IoT scenarios.

Scalability Challenges in IoT Modeling The burgeoning IoT landscape presents a formidable challenge in terms of scalability. As the number of interconnected devices continues to skyrocket, traditional simulation methods struggle to cope with the sheer volume of entities and the complexity of interactions. Scaling up simulations linearly with the increasing number of entities becomes impractical, necessitating novel approaches to modeling and analysis.

The Concept of Multi-Level Simulation Multi-level simulation is a visionary approach designed to address the scalability limitations of traditional simulations. At its core, it consists of hierarchically organized simulation levels, each catering to specific aspects of the IoT scenario. These levels collectively represent a holistic simulation environment that strikes a balance between computational complexity and simulation fidelity.

The key components of this multi-level simulation approach include:

1. **Top-Level Simulation (Level 0):** The top-level simulator orchestrates the overall simulation, setting the stage for the entire IoT scenario. It operates at a coarser level of detail, focusing on high-level behaviors, movement patterns, and service interactions. In the context of S^3 [62] and similar frameworks, this simulator is responsible for managing the entire simulated territory, including smart cities, regions, or shires. Communication between simulated entities is often based on simplified models to reduce computational overhead.

2. **Lower-Level Simulations (Level 1 and Beyond):** Lower-level simulators are domain-specific and responsible for modeling fine-grained interactions, entities, and phenomena within the IoT scenario. They operate at a higher level of detail, capturing intricate behaviors, communication protocols, and movement dynamics. The lower levels can encompass various simulators, such as discrete event simulators (e.g. OMNeT++), agent-based models, or specialized simulators tailored to specific IoT domains. These simulators focus on specific aspects like vehicular traffic, market dynamics, or wireless communication, enhancing the overall fidelity of the simulation.

Advantages of Multi-Level Simulation for IoT Multi-level simulation offers several compelling advantages when applied to IoT modeling:

1. **Scalability:** By distributing the simulation workload across multiple levels, multi-level simulation mitigates scalability issues. The top-level simulator simplifies the overall structure, enabling simulations of vast territories without overwhelming computational resources.

2. **Fidelity and Realism:** Multi-level simulations strike a balance between computational efficiency and realism. Fine-grained lower-level simulations capture intricate details of IoT components, ensuring the accuracy of specific domain behaviors. This approach enables high-fidelity modeling of scenarios involving diverse IoT devices and interactions.

3. **Interoperability:** Multi-level simulation encourages interoperability between different simulation tools and models. Diverse simulators can coexist within the same framework, facilitating the integration of specialized domain knowledge and models. This interoperability is crucial for simulating complex, heterogeneous IoT scenarios involving various technologies and standards.

Challenges and Considerations Despite its advantages, multi-level simulation poses certain challenges and considerations:

1. **Synchronization:** Coordinating interactions between the top-level and lower-level simulators can introduce synchronization complexities. Ensuring that the models at different levels remain coherent and aligned is crucial for accurate simulations.

2. **Data Exchange:** Effective data exchange mechanisms are essential for seamless communication between different simulators. Developing standardized interfaces for data interchange is necessary to maintain consistency across levels.

3. **Approximation Errors:** The use of hybrid simulation approaches, as seen in [64], may introduce approximation errors. Careful validation and calibration of models at different levels are essential to minimize these errors.

Conclusion

Multi-level simulation stands as a promising paradigm for scalable and realistic IoT modeling. By hierarchically organizing simulations and enabling interoperability between diverse simulators, it addresses the challenges posed by the exponential growth of IoT devices and interactions. While challenges related to synchronization, data exchange, and approximation errors persist, ongoing research in this area holds the potential to revolutionize the way we simulate and understand complex IoT scenarios. As the IoT ecosystem continues to expand, multi-level simulation will likely play an increasingly vital role in designing, testing, and optimizing IoT applications and systems.

3.3.2. Leveraging ACOSO and OMNeT++ for Hybrid IoT Simulation

In this section, we delve into an innovative hybrid simulation approach, proposed by Fortino and his team in a series of works [66], [67], and [68]. This approach focuses on the ”thing aspect” of the Internet of Things (IoT), emphasizing the critical role played by Smart Objects within IoT ecosystems. The core concept revolves around tightly coupling these Smart Objects with agents, effectively transforming the IoT into a decentralized multi-agent system.

IoT as a Multi-Agent System

Fortino's approach capitalizes on the principles of Agent-Based Computing. Smart Objects within the IoT are treated as agents, collectively forming a decentralized multi-agent system. This multi-agent perspective enhances interoperability among heterogeneous subsystems and distributed resources. Modeling IoT as a multi-agent system streamlines system development and modeling. It enhances scalability and robustness while reducing design time and time to market. The multi-agent approach inherently aligns with the distributed nature of the IoT.

ACOSO: The Agent-Based Simulator

Fortino et al. employ the ACOSO (Agent-Based COoperating Smart Object) simulator as a central tool for modeling the behavior and interactions of Smart Objects. ACOSO serves as the primary tool for simulating the intricate behaviors of these Smart Objects. To simulate the complex and nuanced communication dynamics between Smart Objects, the researchers harness OMNeT++ in conjunction with the INET framework. OMNeT++ provides a versatile and highly detailed platform for modeling communication and networking aspects.

Experimental Focus on Communication Characteristics

Fortino's experimental endeavors prioritize gaining insights into the communication patterns among Smart Objects within various deployment scenarios. Instead of concentrating on traditional performance metrics, these experiments seek to uncover the inherent nature of interactions within IoT systems. It's important to note that Fortino's approach relies on agent-based modeling, which differs from the discrete event simulation methodology presented in this dissertation. However, both approaches share a common thread in utilizing OMNeT++ to intricately model communication aspects within IoT systems.

Conclusion

Fortino and his research team's innovative hybrid simulation approach, which unites ACOSO and OMNeT++, demonstrates the adaptability and versatility of simulation techniques in the context of IoT research. By treating Smart Objects as agents within a multi-agent system, this approach offers a fresh perspective on IoT modeling, with a strong emphasis on decentralized interactions and interoperability. The synergy between ACOSO's agent-based modeling and OMNeT++'s communication modeling capabilities provides a comprehensive toolset for exploring the dynamic nature of IoT communication. While this approach diverges from the discrete event simulation methodology developed in this dissertation, it underscores the diverse range of simulation techniques available to researchers and practitioners in the ever-evolving field of IoT. As IoT systems continue to grow in complexity, the choice of simulation approach, whether agent-based or discrete event-based, will depend on specific research objectives and system characteristics, further enriching the IoT simulation landscape.

3.3.3. Emulating Large-Scale IoT Scenarios with MAMMotH

In this section, we delve into an innovative approach proposed by Öooga, Ou, Deng, and Ylö-Jääski in [69]. Rather than focusing solely on simulation, their work revolves around the emulation of massive-scale Internet of Things (IoT) scenarios. The choice to emphasize emulation over simulation stems from a crucial distinction – emulators aim to replicate real-world conditions and actual device interactions, avoiding the simplifications often inherent in simulations. Moreover, network simulators may overlook critical issues related to message timing.

Rationale for Emulation

Emulation is favored due to its capacity to replicate complex real-world scenarios with high fidelity. Unlike simulations, emulators ensure precise timing of message exchanges. The authors highlight the limitation of current IoT simulators and emulators in handling large-scale testing with millions of nodes, primarily attributed to scalability challenges. They did a survey of 15 different Internet of Things simulators and emulators. Namely: NS2[70], NS3[71], PDNS[72], GTNetS[73], J-Sim[74], Jist[75], COOJA[76], TOSSIM[77], DSSimulator[78], GlomoSIM[79], OMNeT++[80], SensorSIM[81], SENSE[82], EMULAB[83] and ATEMU[84]. They came to the conclusion, that the landscape of tools primarily directed towards Internet of Things (IoT) research predominantly comprises simulators tailored to assess specific facets of IoT networks. It is noteworthy that a substantial portion of these IoT-focused simulators has, over time, either stagnated without active development or been discontinued. This phenomenon could be attributed to their inherent limitations, which often confine them to addressing only isolated aspects of IoT. In contrast, generic network simulators with IoT extensions, exemplified by NS2 and NS3 and their derivatives, exhibit sustained utility within the research community. Additionally, certain variants, like PDNS and NS3 equipped with MPI support, offer the added capability of distributed simulations, thereby extending their applicability to more complex IoT scenarios.

In the realm of IoT testing, emulators stand as an alternative to simulators, offering a more diverse and practical approach. However, it’s crucial to note that most emulators are constrained to operate within a single virtual machine (VM) and are often limited in terms of the number of nodes they can emulate, with the notable exception of EMULAB. This limitation is primarily due to the resource-intensive nature of emulators. Consequently, the deployment of a multitude of nodes for large-scale IoT emulation on a single VM becomes impractical. Consequently, a glaring gap persists in the field of IoT research tools: the absence of a robust, large-scale emulation platform capable of accommodating the intricacies and intricacies of the IoT landscape. This unmet need underscores the demand for the development of a dedicated platform to facilitate large-scale IoT emulation, a domain ripe for exploration and innovation in the IoT research community.

MAMMotH: A Large-Scale IoT Emulator

The authors introduce ”MAMMotH,” an emulator specifically designed to emulate extensive IoT deployments. MAMMotH’s ambitious goal is to simulate tens of thousands of nodes within a single virtual machine (VM), with an envisioned future capability of scaling up to a staggering 20 million nodes. Linear scalability is a fundamental objective of this emulator architecture. MAMMotH targets three core emulation scenarios:

- Mobile devices connected via GPRS to a central base station.
- Wireless Sensor Networks linked through GPRS to a central base station.
- Constrained devices connected to proxies, which are subsequently linked to a backend infrastructure.

Emulations of Links, Gateways and Nodes

To facilitate link emulation, the authors determined that traffic between nodes on proxies or base stations should adhere to GPRS or 802.15.4 profiles. TCP and UDP traffic simulation is planned through the utilization of NS2-derived models combined with a ”netfilter”-style traffic scheduler. Gateway emulation options include employing EMULAB or employing a Linux Virtual Machine equipped with OpenWRT. Gateways play a pivotal role in connecting the IoT ecosystem to broader networks. Node emulation is achieved through the utilization of existing software that supports the Constrained Application Protocol (CoAP). The authors employed a proprietary Java-based node emulator in conjunction with ”libcoap,” which required modification to utilize threads, overcoming the kernel’s limitations on the number of processes. This innovative approach enabled emulation of up to 10,000 nodes within a single VM, with the primary constraints being the maximum number of threads allowed per kernel and the availability of free UDP ports.

Conclusion

The MAMMotH emulator represents a remarkable departure from traditional IoT simulation techniques. Öooga, Ou, Deng, and Ylö-Jääski’s focus on emulation, as opposed to simulation, underscores the significance of replicating real-world conditions in IoT research. By enabling the emulation of massive-scale IoT scenarios with tens of thousands or even millions of nodes, MAMMotH addresses scalability challenges that have hindered previous simulators and emulators. The emulator’s distinctive approach, including link emulation, gateway emulation, and node emulation, provides a powerful platform for investigating IoT systems’ behavior and performance in real-world-like conditions.

In this landscape of diverse IoT research tools and techniques, MAMMotH stands as a testament to the adaptability and ingenuity of researchers in the field. While this approach differs from the discrete event simulation methodology developed in this dissertation, it broadens the spectrum of available tools for IoT exploration. As IoT systems continue to evolve and grow in complexity, the choice between emulation, simulation, or other techniques will depend on the specific research objectives, further enriching the IoT emulation and simulation ecosystem.

3.3.4. DEUS, COOJA and NS3

This chapter embarks on an exploration of DEUS, COOJA, and NS-3, three pivotal tools in the domain of IoT research and simulation. Together, they offer a comprehensive suite for investigating and understanding complex IoT systems[85]. Below, we delve into the details of each tool, elucidating their unique features, capabilities, and applications.

DEUS: A Discrete Event Urban Simulator

DEUS, short for Discrete Event Urban Simulator, plays a crucial role in the realm of IoT research. This simulator is specially designed to focus on urban environments, making it invaluable for studying IoT deployments within cityscapes.

DEUS is particularly adept at modeling IoT scenarios in urban settings, facilitating the analysis of various IoT applications in smart cities. It operates on a discrete event simulation model, enabling precise control and tracking of events and interactions within the simulated urban environment. DEUS offers a detailed, granular representation of urban landscapes, allowing researchers to evaluate IoT systems' performance in real-world urban conditions. The simulator's emphasis on urban scenarios makes it an ideal choice for understanding the intricacies of IoT deployments in densely populated areas, where factors like mobility, communication, and resource management are of paramount importance.

COOJA: A Network Simulator for IoT

COOJA is a prominent network simulator tailored explicitly for IoT research. Its primary focus is on simulating wireless sensor networks, making it an invaluable tool for exploring IoT applications that rely on low-power, wireless communication.

COOJA specializes in modeling wireless sensor networks, making it particularly well-suited for IoT scenarios involving resource-constrained devices and low-power communication protocols. This simulator provides a highly detailed representation of sensor nodes and their interactions, allowing researchers to assess the performance and behavior of IoT systems with a high degree of precision. COOJA's support for Contiki, a popular operating system for IoT devices, further enhances its relevance in IoT research. Researchers can simulate Contiki-based IoT deployments with ease. The simulator's focus on network-level modeling and wireless communication makes it an indispensable tool for investigating IoT applications that rely on efficient data transmission and network management.

NS-3: A Versatile Network Simulator

NS-3, or Network Simulator 3, is a versatile and widely-used network simulator that extends its capabilities to encompass IoT research. With a broad range of features and extensive community support, NS-3 is a go-to choice for simulating diverse networking scenarios, including those within the IoT domain.

NS-3 boasts versatility, enabling researchers to model a wide spectrum of networking scenarios, from traditional computer networks to emerging IoT deployments. Its modular architecture and support for various communication protocols make it adaptable to different IoT use cases, allowing researchers to explore IoT applications across diverse domains. NS-3's active community and extensive documentation ensure a wealth of resources for users, making it accessible and user-friendly, particularly for newcomers to IoT simulation. Researchers can leverage NS-3's features to simulate IoT scenarios involving diverse devices, communication technologies, and network topologies, offering a broad canvas for IoT exploration.

Conclusion

DEUS, COOJA, and NS-3 collectively provide a comprehensive toolkit for IoT research and simulation, each offering unique strengths and capabilities. DEUS excels in urban IoT scenarios, offering a fine-grained representation of city environments. COOJA specializes in wireless sensor networks, making it indispensable for IoT applications reliant on low-power, wireless communication. NS-3, with its versatility and extensive community support, accommodates a wide range of IoT use cases, ensuring researchers have access to a flexible and powerful simulation platform.

As the IoT landscape continues to evolve, the choice of simulation tool depends on specific research objectives and the nature of the IoT deployment under investigation. Whether it's urban environments, resource-constrained devices, or diverse networking scenarios, these tools empower researchers to delve into the complexities of IoT systems, paving the way for advancements in this dynamic field.

3.4. Comparison of IoT Simulation Approaches

Comparing the afore mentioned various IoT simulation approaches and the simulator developed in this dissertation, reveals a spectrum of strategies and capabilities tailored to different research needs and scenarios. Each approach possesses distinct advantages and trade-offs, contributing to the diversity of tools available for IoT research. Below, I conduct a comparative analysis to shed light on their relative merits and applications.

3.4.1. Interconnected Simulation Levels

By tightly integrating several simulation levels, encasing domain-specific simulators inside models, and coordinating them in a single event-loop, the suggested simulator sets itself apart. This approach fosters a seamless exchange of information between simulation levels and allows for flexible model interchange. Unlike some other simulators, the boundaries between hierarchical levels are more fluid, permitting dynamic model swapping. This adaptability proves advantageous when dealing with IoT scenarios that demand varied levels of detail and continuous-time approximations.

3.4.2. Versatility and Diversity

DEUS, COOJA, and NS-3 cater to diverse IoT research needs by offering a range of specialized features. DEUS excels in urban environments, providing a granular representation of cityscapes. COOJA shines in modeling wireless sensor networks, making it indispensable for low-power IoT applications. NS-3's adaptability and extensive protocol support accommodate a wide spectrum of IoT use cases. The choice among these tools depends on the specific research objectives and the nature of the IoT deployment.

3.4.3. Community Support and Documentation

NS-3 stands out for its active community and extensive documentation, making it an attractive choice for researchers, especially those new to IoT simulation. Access to a wealth of resources and user-friendly features ensures a smoother learning curve and efficient problem-solving.

3.4.4. Synchronization and Scalability

Synchronization mechanisms and scalability differ across the approaches. The interconnected model approach offers a flexible solution for accommodating different simulation models and time-steps, enabling efficient synchronization without complex processes. On the other hand, some approaches, like the combined use of S^3 and OMNeT++, necessitate meticulous synchronization strategies and may face scalability constraints, particularly in scenarios with a large number of entities.

3.4.5. Continuous-Time Approximations

The proposed simulator's capability to accommodate continuous-time approximating techniques without intricate synchronization processes sets it apart. This flexibility enables the integration of diverse model types within the simulation, providing researchers with the freedom to explore various modeling approaches seamlessly.

3.4.6. Conclusion

In conclusion, the comparison of these IoT simulation approaches underscores the importance of choosing the right tool for the specific research context. Each simulator offers unique strengths and caters to distinct IoT scenarios. Researchers must consider factors such as the complexity of the simulation, the level of detail required, and the nature of the IoT deployment when selecting the most suitable tool. The diversity in available options reflects the multifaceted nature of IoT research, with each simulator contributing to advancements in this dynamic field. Ultimately, the effectiveness of an IoT simulation approach hinges on its alignment with the research objectives and the intricacies of the IoT system under investigation.

Chapter 4

Simulation of IoT networks

Contents

Chapter 4 delves into the intricate world of simulating IoT networks as the underlying concepts of my work. At its core, this chapter unfolds the fundamental concept underpinning IoT network simulation, scrutinizing the pivotal requirements and specifying Discrete Event Systems. It navigates through the rigorous realm of Discrete Event Specification, embracing the DEVS formalism, atomic DEVS, network DEVS, closure under coupling, simulation loops, parallel DEVS, and the integration of Continuous Time Hybrid DEVS. The chapter then transitions to the practical domain of OMNeT++, dissecting its implementation for models, message passing, the simulator core, and the simulatinon environment. With a keen eye on the simulation approach, it illuminates the importance of level hierarchy, model reuse, and synchronization. Implementing the simulation core takes center stage, exploring the core simulation framework, the hierarchy middleware, and seamless OMNeT++ integration. Additionally, the chapter delves into the critical dimensions of Hardware in the Loop, complete with OMNeT++ socket servers, VICINITY Adapters, and Simulation Schedulers. The chapter concludes with a comprehensive exploration of the dynamic realm of Human in the Loop, offering an extensive conceptual understanding of these intricate facets of IoT network simulation. Please note that certain portions of the content and findings presented in this chapter have been previously disseminated in conferences such as [86, 87, 88, 89, 90], and have been thoughtfully incorporated into this dissertation to provide a comprehensive and coherent perspective on the subject matter.

4.1. Introduction

The discrete-event network simulation framework Omnet++ has demonstrated its remarkable capabilities in various contexts. However, when tasked with the simulation of an entire smart city, it encounters notable performance challenges. These issues become particularly pronounced when simulating the intricate network of models required to represent a comprehensive urban environment. Nevertheless, the potential benefits of simulating an entire smart city, including the complex interactions between its constituent components, hold immense promise for advancing the concepts of smart cities and the Internet of Things (IoT).

This approach endeavors to harness the formidable capabilities of the Omnet++ framework, renowned for its prowess in simulating network traffic, and as a foundation for INET in simulating internet-related scenarios. To mitigate the performance bottlenecks encountered when simulating vast urban environments, we introduce a lightweight custom simulation framework. The overarching objective is to seamlessly integrate the strengths of Omnet++ while addressing its performance limitations.

To achieve this goal, we employ the principles of hierarchical modeling of

Discrete-event Systems (DEVS) models to establish a ”time hierarchy” within the simulation. The fundamental concept involves dynamic transitions between models that provide simplified representations of substantial city segments (e.g. city quarters or districts) with relatively coarse-grained time resolutions. Concurrently, we incorporate interconnected networks of models to capture finer-grained details within specific areas of interest. This dynamic approach empowers us to economize computational resources by utilizing coarser abstractions for less critical portions of the simulated city, while finely tuning our focus to observe crucial details precisely when they become relevant.

The proposed simulation framework excels in supporting multi-level simulations while relying exclusively on a discrete-event simulation technique. By enabling dynamic model switching across various levels of abstraction, we effectively simplify extensive sections of a simulated IoT network with a coarser time resolution. This adaptability allows us to dynamically scrutinize and prioritize details pertinent to specific simulation scenarios, providing a nuanced perspective on the intricacies of smart city operations.

4.2. Concept

The primary objective of this approach is to formulate a versatile DEVS (Discrete-Event System Specification) simulator that possesses hybrid simulation capabilities tailored to accommodate large-scale IoT and Smart City simulations, specifically designed to address the unique requirements outlined within the context of the VICINITY project. As elucidated in Section 4.2.1, we must consider both the general requisites of IoT simulations and the specialized demands inherent to the VICINITY project’s use cases.

One key advantage of this approach is the consolidation of simulation tools into a singular, comprehensive simulator, obviating the need to orchestrate a disparate array of simulators for each unique scenario. This consolidation promises enhanced efficiency and simplicity in managing complex simulations.

The core architectural concept revolves around adopting a time-hybrid approach to address the intricacies associated with handling an extensive multitude of models and events within large-scale simulations. This approach is pivotal in achieving the envisioned performance improvements and scalability required for these simulations. The emphasis is on meticulously managing the interconnections that constitute the crux of IoT systems. Consequently, our model is centered around these connections, with a network simulator serving as its core.

When considering the choice between DEVS and DETS (Discrete-Event Time-Stepped), we opt for DEVS, primarily because Omnet++ itself adheres to the DEVS framework. While integrating DETS into Omnet++ would be feasible, it introduces unnecessary overhead due to the presence of empty time frames. Nevertheless, it is important to note that DETS inherently offers a more straightforward implementation of changes in time hierarchy. By merely adjusting a dynamically calculated scaling factor, DETS allows for the adaptation of time steps across various simulation levels, enhancing flexibility.

Our architectural design adopts a tree-like structure to organize the time hierarchy, characterized by multiple simulators operating in tandem, rather than relying on a single monolithic simulator or one dedicated simulator per level. This design closely aligns with the principles of DynDEVS (Dynamic Discrete-Event System Specification), a fundamental concept worth mentioning in the context of our approach.

Central to our approach is the fusion of Omnet++ (augmented by INET) with a general-purpose DEVS simulator. This integration enables the translation of agent-based models into the DEVS framework, with a primary focus on enhancing scalability. We achieve this scalability through the introduction of a time hierarchy, which introduces finer-grained time steps as we delve deeper into the simulation. This innovative approach empowers us to efficiently manage the intricacies of large-scale IoT and Smart City simulations, facilitating a higher degree of precision and control.

4.2.1. Requirements

The requirements for a comprehensive IoT simulator, drawn from existing works in the field (as highlighted in Section 3), along with specific demands emerging from the VICINITY project, are multifaceted and crucial in shaping the framework's architecture. These requirements encompass a range of essential facets:

1. **Simulating Thousands of Interconnected Devices:** The simulator must be capable of efficiently simulating a vast number of interconnected devices. While certain IoT scenarios, such as smart homes, involve a relatively limited number of devices, larger-scale applications, like city-wide services, necessitate the simulation of thousands of entities. Scalability is key to providing valuable insights across a spectrum of scenarios [62].

2. **Real-Time or Near Real-Time Simulation:** The simulator should offer the ability to run in (almost) real-time, especially for proactive approaches. While detailed simulations are valuable for understanding fine-grained processes, large-scale IoT scenarios in smart cities require agility. The framework should incorporate techniques to enable these approaches without sacrificing real-time performance.

3. **Hardware in the Loop (HIL) Capabilities:** To facilitate the analysis of prototypes and real-world device behavior, the simulation framework should ideally support hardware in the loop (HIL) capabilities. Alternatively, it should provide interfaces for seamless integration with existing HIL solutions.

4. **High Scalability:** In conjunction with the first two requirements, the framework must exhibit high scalability. Real-time capabilities should not diminish when simulating scenarios involving thousands of interacting entities.

5. **Parallel and Distributed Simulation:** The framework should either include built-in support for parallel and distributed simulations or be designed in a manner that facilitates the incorporation of these capabilities. This is in alignment with findings from [64].

6. **Rapid Model Development:** To meet the project's time-sensitive requirements, the simulator should enable rapid model development. Ideally, it should support the use of functional mock-up interfaces and streamline the process of creating complex models.

7. **Integration of Heterogeneous Technologies:** The simulator should have the capability to unify various heterogeneous technologies at all levels of IoT. It should be able to model and simulate the diverse technologies relevant to different IoT domains and enable seamless interaction between them.

8. **Support for Domain-Specific Simulators:** As a final requirement, the framework should facilitate the integration of additional domain-specific simulators. If, during the framework's deployment, the need arises to incorporate specialized simulators, this integration should be straightforward and require minimal effort.

These requirements serve as the foundational pillars upon which the proposed IoT simulation framework will be constructed. Each requirement represents a vital aspect essential for the simulator's success in addressing the challenges posed by large-scale IoT and Smart City simulations, particularly within the context of the VICINITY project.

4.2.2. Specification of Discrete Event Systems

Discrete Event Systems (DEVS), exemplified by computer networks, Very Large-Scale Integration (VLSI) circuits, and even activities as nuanced as a ping-pong game, are characterized by their inherent nonlinearity. This nonlinearity arises from the amalgamation of two key attributes: the system's current state and the state transition it undergoes [60]. Consequently, the modeling of discrete systems demands specialized techniques.

Throughout the history of mathematics, several formalisms have emerged to address the unique characteristics of discrete event systems. These formalisms encompass queuing theory, Petri nets, Markov processes, and Finite State Machines (FSM). However, in 1975, Bernard Zeigler embarked on a mission to establish a unified foundation that would seamlessly integrate various mathematical modeling techniques with the complexities of highly nonlinear system simulation [91].

The crux of the challenge in modeling discrete event systems, as elucidated in Zeigler's seminal work "Theory of Modeling and Simulation," lies in the specification of state changes and the deterministic sequences governing them. Transitions between states are contingent upon the antecedent sequential state, predicated on inputs or outputs [91]. Each discrete step within these scenarios is aptly termed an event, and events can also be triggered by the elapsing of a designated time interval.

Bernard Zeigler's formalism can be viewed as an extension of the Moore Machine formalism, essentially representing a Finite State Machine (FSM). This extension introduces two pivotal enhancements: the concept of a "life-span" for each state and the incorporation of "hierarchical coupling." These additions empower the formalism to accommodate the intricate dynamics and intricacies of highly nonlinear systems, thus rendering it an invaluable tool for modeling and simulating complex discrete event systems.

For example an atomic model is specified as:

$$M =< X, Y, S, t_a, \delta_{ext}, \delta_{int}, \lambda > \tag{4.1}$$

where:

X	set of input events
Y	set of output events
S	set of sequential states
$\delta_{ext} : Q \times X \to S$	external state transition function
$\delta_{int} : S \to S$	internal state transition function
$\lambda : S \to Y$	output function
$t_a : S \to R_0^+ \cup \infty$	time advance function

4.3. Discrete Event Specification

The Discrete Event Systems (DEVS) formalism is an extension of the Finite State Automata formalism, enriched with concepts from Discrete Event Simulation (DES)[92]. Introduced by Zeigler in 1976[91], DEVS was conceived as a unifying framework for discrete event modeling and simulation. In this section, we present an overview of the DEVS formalism, with a focus on both atomic and coupled DEVS formalism.

4.3.1. DEVS Formalism

DEVS is specifically designed for modeling Discrete Event Systems. It offers mechanisms to describe systems whose states change through deterministic transitions between sequential states or due to external inputs and their corresponding outputs.

In DEVS, changes in the system state are referred to as events, which can arise from various sources. These events may result from the expiration of a time interval, external inputs, or the generation of model outputs. DEVS categorizes events into two types: internal events and external events. How the discrete event system responds to these events is further elucidated in Section 4.3.2.

To manage the complexity of such systems, DEVS employs a hierarchical structure composed of simpler, coupled components[60]. These coupled components can take the form of atomic DEVS models or coupled network

DEVS models, which themselves can include coupled models. The connections, or couplings, between these models dictate how they interact with each other. For instance, one model's output can serve as the external input for another model. DEVS facilitates hierarchical modeling by being closed under coupling[92]. This property ensures that for every such network, a resultant atomic model can be computed, behaving equivalently to the network. Consequently, DEVS supports the creation of complex hierarchical models.

The subsequent sections will provide in-depth explanations of atomic, network, and closure under coupling concepts.

4.3.2. Atomic DEVS

The Atomic DEVS model is defined as:

$$atomic \equiv \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle, \tag{4.2}$$

Here's a detailed breakdown of its components:

- **Sequential States (S)**: These states represent the discrete states of the system. Typically, S is a structured set $S = \times_{i=1}^{n} S_i$, which formalizes multiple concurrent aspects of a system[92].
- **Time Advance Function (ta)**: The ta function maps each sequential state $s \in S$ to a positive real number, indicating the time the system remains in that state before an internal event occurs, and a transition to the next state takes place.
- **Internal Transition Function (δ_{int})**: This function defines how the system transitions from one sequential state to another within the given time frame.
- **Output Set (Y)**: The output set represents the possible outputs produced during an internal transition. Typically, Y is a structured set $Y = \times_{i=1}^{l} Y_i$[92].
- **Output Function (λ)**: The λ function maps each internal state to an element in the output set Y or the empty set $\varnothing$. Outputs are generated only during internal transitions.

To model external inputs from the input set X, which is also typically structured[92], the concept of total states (Q) is introduced. These total states incorporate both the sequential state s and the elapsed time e since the last state transition:

$$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$$

The elapsed time e can be used to calculate the time left in the current state as $\sigma = ta(s) - e$.

To handle input events ($x \in X$) occurring within a bounded time interval, the external transition function (δ_{ext}) is employed:

$$\delta_{ext} : Q \times X \rightarrow S$$

In summary, in an atomic DEVS model, the system is in a sequential state $s \in S$ at any given time. If no external events occur, it remains in this state until the time advance function $ta(s)$ expires. Upon expiration, the output function $\lambda(s)$ produces an output $y \in Y$, and the model transitions to a new state as determined by the internal transition function $\delta_{int}(s)$. This is referred to as an internal transition. A state with $ta(s) = 0$ is called a transient state, and a state with $ta(s) = \infty$ is termed a passive state. When an external event in the form of external input $x \in X$ occurs, an external transition is triggered via the external transition function $\delta_{ext}(s, e, x)$, where the current state s and elapsed time e define the current total state.

4.3.3. Network DEVS

A Network DEVS model, also known as a coupled DEVS model, is defined as a network composed of interconnected components:

$$network \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle, \tag{4.3}$$

Here is a breakdown of the components within this network:

- Input and Output Sets for Self (X_{self} and Y_{self}): These sets define the allowed inputs and outputs for the coupled model itself.
- Set of Unique Components (D): D represents a set of unique components within the model. It is important to note that the model itself ($self$) is not included in D. Components in D are denoted as $M_i | i \in D$, and each of these components must be an atomic DEVS model.
- Influencees (I_i): I_i represents the set of components influenced by $i \in D \cup self$. It describes how the components within the network are coupled.

The network adheres to certain rules:

- **Hierarchical Principle**: No component within the network is allowed to influence other components outside of the coupled network model.
- **No Self-Influence**: Components cannot influence themselves to prevent infinite loops in the model's state.

These rules are expressed as:

$$\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$$

and

$$\forall i \in D \cup \{self\} : i \notin I_i.$$

To complete the coupling of components, the model employs output-to-input translation functions $Z_{i,j}$:

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\},$$

$$Z_{self,j} : X_{self} \rightarrow X_j, \forall j \in D,$$
$$Z_{i,self} : Y_i \rightarrow Y_{self}, \forall i \in D,$$
$$Z_{i,j} : Y_i \rightarrow Y_j, \forall i, j \in D.$$

These functions enable the translation of outputs and inputs between components:

- $Z_{self,j}$ translates outputs from the self component to inputs of component j.
- $Z_{i,self}$ translates outputs from component i to inputs of the self component.
- $Z_{i,j}$ translates outputs from component i to inputs of component j.

Within such coupled models, it is possible that multiple state transitions of the components occur simultaneously due to the discrete-event abstraction. To address these situations, a tie-breaking function *select* is included in the formalism. This function selects a unique component from a non-empty subset E of D to determine the order in which components are handled.

In summary, a Network DEVS system does not explicitly model a state but rather implicitly represents its state and behavior through interconnected atomic DEVS models. These models are coupled together into a network, and the interactions between them are defined by input and output translation functions, allowing for the modeling of complex systems with multiple components.

4.3.4. Closure under coupling

The concept of Closure under Coupling allows the interchange of a DEVS network model with an equivalent atomic one. This interchange is achieved by constructing a resultant atomic DEVS model. The significance of this principle is that it relaxes the previous constraint, which stated that a network DEVS model could only consist of atomic components. This idea will be applied in this dissertation to realize a time hierarchy. To illustrate how this works, we will delve into the construction of the resultant atomic model based on the explanation in [92].

Starting from the network DEVS definition in Equation 4.3, which encompasses all the model's components:

$$M_i = \langle S_i, X_i, Y_i, \delta_{int,i}, \delta_{ext,i}, \lambda_i, ta_i \rangle \forall i \in D,$$

a resultant atomic DEVS model is constructed following Equation 4.2. To do this, we first build the set of total states for all components:

$$Q_i = \{(s_i, e_i) | s \in S_i, 0 \leq e_i \leq ta_i(s_i)\} \forall i \in D,$$

Next, we create the set of all sequential states of the resultant model through a product:

$$S = \times_{i \in D} Q_i.$$

The time advance function of the resultant atomic model selects the most imminent event time among all coupled components as its output. Imminent events are those events scheduled to occur next. The function picks the most immediate event and moves the clock forward in the smallest increments until all of the components undergo an internal transition:

$$ta(s) = min\{\sigma_i = ta_i(s_i) - e_i | i \in D\}.$$

This time advance function allows us to create the IMM set of imminent components that are planned for a simultaneous internal transition:

$$IMM(s) = \{i \in D | \sigma_i = ta(s)\}.$$

From this set, we choose one component i^* using the select function mentioned in the previous subsection (Section 4.3.3). Now, the output of this component is computed before the internal transition as follows:

$$\lambda(s) = Z_{i^*,self}(\lambda_{i^*}(s_{i^*}), if self \in I_{i^*}).$$

The following changes occur in the various components of the overall state due to the internal transition function:

$$\delta_{int} = (\dots, (s'_j, e'_j), \dots),$$

where

$$(s'_j, e'_j) = \begin{cases} (\delta_{int,j}(s_j), 0), & for j = i^* \\ (\delta_{ext,j}(s_j, e_j + ta(s), Z_{i^*,j}(\lambda_{i^*}(s_{i^*}))), 0), & for j \in I_{i^*}, \\ (s_j, e_j + ta(s)), & otherwise \end{cases}$$

In this transformation, the component i^* transitions to sequential state $\delta_{int,i^*}(s_{i^*})$, the elapsed time e is reset to zero, and all influencees of i^* change their state as an external transition occurs when the output-to-input translated output of i^* arrives at them with an elapsed time equal to the time advance $ta(s)$. Subsequently, their elapsed time is also set to zero. The state of all other components remains unchanged.

When an external event occurs, the external transition function transforms the total state as follows:

$$\delta_{ext}(s, e, x) = (\dots, (s'_i, e'_i), \dots),$$

where

$$(s'_i, e'_i) = \begin{cases} (\delta_{ext,i}(s_i, e_i + e, Z_{self,i}(x)), 0), & for i \in I_{self} \\ (s_i, e_i + e), & otherwise \end{cases}$$

This process ensures that the resultant atomic model captures the behavior of the entire network DEVS model, allowing for the interchange between the two representations.

4.3.5. The simulation loop

The implementation of an abstract simulator in the context of DEVS formalisms, including atomic and network DEVS, and closure under coupling, is facilitated through a simulation loop.

In line with the hierarchical design principle of DEVS models, this implementation employs various execution engines, referred to as processors, throughout the model hierarchy[60]. These processors include:

1. **Atomic Simulator**: Associated with an atomic model, this simulator controls its internal and external state change functions, manages the output function, and keeps track of the last and next event times for this model.
2. **Coordinator**: Linked to a coupled model, the coordinator's role is to route and translate input and output between the coupled components and the external environment.
3. **Root Simulator**: Responsible for coordinating the global aspects of the simulation.

The simulation loop operates by passing messages, which convey information about the source or target of an event, the event time, and possibly a value to be transported. Four types of messages are used:

- * messages: Represent internal events.
- X messages: Carry information about external inputs.
- Y messages: Transport a model's output.
- done messages: Indicate that a model has completed its task.

The simulation cycle begins with the root coordinator examining its list of external input events and the scheduled times for the next internal events. The model with the smallest remaining time until the next internal event is referred to as an imminent model, and its associated simulator is the imminent simulator. Depending on whether the next event is an external input or an internally scheduled event, the root simulator generates a * or X message, respectively.

If a * message is created, it is propagated recursively by the coordinators, starting with the top-level coordinator and then passed down the hierarchy to reach the imminent simulator. The imminent simulator first executes its output function λ. If λ generates an output other than the non-event $\varnothing$, a Y message is created and sent to the parent coordinator. The coordinator then executes its $Z_{i,j}$ function to translate the output message into an X message and forwards it to the destination coordinator linked to this model. Subsequently, the simulator executes the internal transition function δ_{int} to update the model's state. Afterward, the time advance function $ta(s)$ is executed to schedule the next internal event. The time in the future when this event will occur is then sent in a done message to the parent coordinator. When the coordinator receives done messages from all its children, it selects the one with the earliest future time and forwards it to its coordinator. Once the root simulator has received done messages from all its children, it updates its next event time, and the cycle begins anew.

If, on the other hand, the root simulator chooses an external input and generates an X message, this message traverses all coordinators along its path to reach the corresponding simulator. The atomic simulator then executes the external transition function δ_{ext} and subsequently the time advance function. Finally, a done message is generated, which serves to schedule further future events[60],[92].

4.3.6. Parallel DEVS

In the previous section (Section 4.3.3), we introduced the network DEVS formalism, where the select function determined the sequence of execution when two models were scheduled for state transitions simultaneously. This led to the sequential execution of imminent models by a coordinator, which posed challenges in representing systems with coincident state changes.

To address this issue, Parallel DEVS was introduced as an extension to the DEVS formalism. Parallel DEVS enables models to receive multiple external events simultaneously and process them in a single step. In an atomic Parallel DEVS model, the components are defined as:

$$atomic \equiv \langle S, X, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle,$$

Here, S, X, Y, δ_{int}, δ_{ext}, λ, and ta remain the same as in the standard DEVS atomic model (as defined in equation 4.2). However, there's an additional function called δ_{con}, which is the confluent transition function:

$$\delta_{con} : S \times X \rightarrow S$$

The elapsed time ($e = ta(s)$) equals the outcome of the time advance function, making this function simply a particular case of the external transition function. When an internal and external event take place at the same time, it is used to determine the new state of the model[60].

In addition to the adapted definition of atomic models, atomic models in Parallel DEVS maintain external input events in multi-sets, often referred to as Bags[60]. During each iteration of the simulation loop, when a model generates output events, they are first routed to their destination model and stored in the bag. Afterward, the new states of the models are determined by one of the three state transition functions: δ_{int}, δ_{ext}, or δ_{con}.

For network models in Parallel DEVS, the only notable change is the omission of the select function. The remaining functionality remains largely unchanged[93]. This extension allows for more efficient modeling of systems with coincident state changes.

4.3.7. Continuous Time Hybrid DEVS

DEVS, as previously discussed, has a significant limitation in that it cannot directly simulate continuous-time models. To address this limitation, a concept called hybrid systems is introduced. Hybrid systems aim to approximate continuous models using discrete event models, effectively integrating discrete events and numerical integration steps. In this context, an atomic discrete event model can encapsulate and approximate a continuous model in its state space form, provided it has methods for event detection and numerical integration. Interactions with such models can only occur through discrete events.

A hybrid model combines continuous state variables $x \in \mathbb{R}^m$ with a set of discrete state variables Q. Transitions in hybrid systems depend on both continuous and discrete state variables. The time advance function, which determines these transitions, is defined as:

$$ta((x_k, q)) = G((x_k, q))$$

Here, x_k represents the value of x at time t_k. The function $G : (\mathbb{R}^m \times Q) \rightarrow \mathbb{R}_0^\infty$ calculates the time until the next event occurs.

Hybrid systems can change their state in response to internal events at $ta((x_k, q)) = 0$ or in response to external events at $h \leq G((x_k, q))$. The value of x at time t_{k+1} can be computed as:

$$x_{k+1} = x_k + \int_{t_k}^{t_k+h} f(x(t), q) dt,$$

Here, $f(x, q)$ represents the function that governs the continuous evolution of x. The model's responses to events are defined as follows:

- $\hat{\delta}_{int}$: Response to internal events.
- $\hat{\delta}_{ext}$: Response to external events.
- $\hat{\delta}_{con}$: Response to concurrent (continuous) events.
- $\hat{\lambda}$: Output produced in response to events.

These responses are defined based on the hybrid system's continuous state and discrete state variables. The relationship between these hybrid responses and the standard DEVS responses for internal, external, and concurrent events is established.

Events in hybrid systems are categorized as time events if G can be directly calculated from the model's state, and as state events if they must be computed from event surfaces in the state space. Event surfaces are constructed using threshold functions $g_p(x(t_k + h), q) = 0$. $G(x(t_k), q)$ can be implicitly defined as the smallest non-negative h that satisfies at least one of these threshold functions.

Simulating hybrid systems requires two additional functions: a numerical integration function and a root-finding function to locate events between integration steps.

In summary, continuous time hybrid DEVS extends DEVS to handle continuous-time models by approximating them using discrete events and numerical integration methods, allowing for the simulation of systems with both continuous and discrete dynamics[93].

4.4. OMNeT++

OMNeT++ is a powerful object-oriented modular discrete event network simulation framework primarily written in C++. It's worth noting that an extended version of OMNeT++ designed for commercial use is known as OMNEST[1]. OMNeT++ is widely adopted within the academic community due to its highly modular architecture and is the foundation for various open-source model suites, with INET being one of the prominent examples[2].

Key Features of OMNeT++ are:

1. **Modularity**: OMNeT++'s modular design allows users to build and extend simulation models with ease. It's structured in a way that promotes code reusability and adaptability.
2. **Discrete Event Simulation (DES)**: OMNeT++ follows the principles of Discrete Event Simulation (DES), making it suitable for modeling and simulating systems where events occur at distinct points in time.
3. **Community and Ecosystem**: The academic community has developed a wealth of open-source model libraries and frameworks that can be seamlessly integrated with OMNeT++, thus expanding its capabilities.

OMNeT++ finds applications in various domains, including computer networks, communication systems, and distributed systems. Researchers and engineers use OMNeT++ to simulate and analyze the behavior of complex systems under different conditions and configurations.

In the upcoming subsections, we'll delve deeper into the implementation principles of Discrete Event Simulation, which underlie OMNeT++.

4.4.1. Implementation of Models

In OMNeT++, models are implemented using modules, which can be categorized into two main types: simple modules and compound modules.

Simple Modules These are analogous to atomic DEVS models. Simple modules are active components of the simulation and represent the fundamental building blocks. They can generate events, process messages, and interact with other modules. In the context of DEVS, they are akin to atomic models. Simple modules have input and output gates that enable communication with other modules.

[1] `https://www.omnest.com/`
[2] `https://inet.omnetpp.org/`

Compound Modules Compound modules are equivalent to network DEVS models. They serve as containers for organizing and connecting multiple simple modules. In DEVS terms, they correspond to systems of interconnected atomic models. Compound modules do not have a direct role in generating or processing events but are crucial for defining the structure of the simulation. The top-level compound module is referred to as the ”system module.”

Module Communication Modules in OMNeT++ communicate through input and output gates. These gates allow messages to be exchanged between modules, enabling the modeling of relationships and interactions within the simulation. Connections between gates define how modules are interconnected. Importantly, connections cannot span multiple hierarchy levels, ensuring a clear and hierarchical structure in the simulation.

Network Description Language (NED) To define the hierarchy and topology of modules, OMNeT++ uses the Network Description Language (NED). NED files store information about the structure of the simulation, specifying the types of modules, their connections, and their properties. NED files work in conjunction with message definitions and C++ source files to form the complete network model.

In summary, OMNeT++ employs a modular approach to model implementation, where simple modules represent active components, compound modules define the structure, and NED files describe the hierarchy and connections between modules. This architecture allows for flexibility and scalability in creating complex network simulations.

4.4.2. Message Passing

In OMNeT++, modules communicate through messages instead of events. These messages, although implemented as an internal subclass of OMNeT’s event system, provide a flexible way to exchange data among modules. Key aspects of message passing in OMNeT++ include:

1. **Message Format**: Messages in OMNeT++ can contain complex data structures. Users have the flexibility to define custom message formats by extending C++ source files. This allows for the representation of various types of information within messages.
2. **Links and Connections**: Messages are transmitted between modules through connections known as ”links.” These connections support several attributes, including:
   - Data Rate: The rate at which data can be transmitted.
   - Propagation Delay: The time taken for a message to travel from sender to receiver.
   - Bit Error Rate: The likelihood of errors occurring in the transmitted bits.

- Packet Error Rate: The likelihood of errors occurring in entire packets of data.

These attributes are typically associated with "channel" objects, which define the characteristics of the link.

3. **Direct Messaging**: OMNeT++ allows modules to send messages directly to each other, bypassing the need for explicit connections. This direct messaging capability enables flexible communication patterns.

4. **Time Advancement**: When a module receives a message, the local simulation time is advanced. This time advancement mechanism ensures that messages are processed in the correct temporal order.

5. **Self Messages**: OMNeT++ supports self-messages, which enable modules to implement autonomous actions. Modules can schedule and receive messages from themselves, allowing for the execution of specific tasks or behaviors independently of external events.

In summary, OMNeT++ employs a message passing mechanism for inter-module communication. Messages can carry diverse data structures, and communication can occur through both explicit links and direct messaging. The simulation time is managed to ensure the correct sequencing of events, and self-messages enable autonomous module behavior.

4.4.3. Implementation of the Simulator

In OMNeT++, the implementation of the core simulation loop (refer to section 4.3.5) involves the orchestration of several pivotal classes, each contributing to the management of events and their subsequent execution. These classes encompass:

Simulator The central role of the Simulator class is to govern the overarching simulation process. It serves as the conductor, directing the execution of events, regulating the simulation's temporal progression, and ensuring the ordered processing of events. The Simulator class closely collaborates with other integral constituents of the simulation loop to maintain temporal synchronization and control.

Future Event Schedule (FES) The Future Event Schedule serves as a foundational data structure responsible for the meticulous tracking of all scheduled events within the simulation. It maintains a comprehensive registry of events, encompassing their temporal stamps and the originating modules. Managed by the Simulator, the FES stands as the arbiter of event chronology, guaranteeing that events are executed in precise temporal succession.

Scheduler The Scheduler component assumes a pivotal role in the formulation of the event execution sequence. It furnishes scheduling strategies that dictate the manner in which events are selected from the FES for execution. OMNeT++ extends the basic scheduling options, encompassing strategies like sequential and real-time approaches, while also accommodating user-defined scheduling strategies for customization.

Collectively, these integrated components within OMNeT++ culminate in a robust simulation framework, adept at addressing a spectrum of simulation scenarios. The Simulator assumes the mantle of event scheduling and execution, ensuring the seamless advancement of simulations. Moreover, the presence of diverse scheduling strategies, including opportunities for users to craft bespoke strategies, bestows flexibility and adaptability upon simulations, aligning them with specific research requisites and objectives.

4.4.4. Implementation of the Environment

In OMNeT++, the environment in which the simulation is conducted is a critical component that encapsulates and extends the functionality of the simulation kernel. OMNeT++ offers users three distinct off-the-shelf environments, each designed to accommodate various simulation scenarios. These environments are responsible for overseeing the execution of the simulation loop, capturing relevant information generated during the simulation, and providing a user-friendly interface for interacting with the simulation.

Of the three available environments, two are graphical environments tailored for visual interaction with the simulation, while the third is a lightweight, command-line-based implementation. For the context of this work, which emphasizes performance in the context of large-scale IoT scenarios, the command-line environment assumes paramount significance. This environment, although minimalistic in terms of user interface, remains instrumental in gathering essential data from the simulation.

Notably, while the graphical environments offer rich visual feedback and interaction capabilities, it's imperative to acknowledge that they can potentially introduce non-negligible overhead and influence the temporal dynamics of the simulation. This characteristic renders them less suitable for applications where precise control over time advancement and performance optimization is of paramount importance.

In summary, OMNeT++ affords users the flexibility to select an environment that aligns with their specific simulation objectives, whether they prioritize visual interactivity or performance optimization. For scenarios demanding fine-grained control over simulation parameters, the lightweight command-line environment emerges as a pragmatic choice, effectively facilitating data gathering and analysis while minimizing extraneous influences on simulation dynamics.

4.5. Approach

The chosen approach for developing the IoT simulation framework builds upon decades of research in discrete event systems, which have evolved a rich set of techniques for parallel and distributed simulation. This well-established foundation includes various extensions to the original specification, such as PDEVS and DynDEVS, that cater to specific problems and domains, including the ability to dynamically change connections within DEVS models [60]. Moreover, the modeling techniques for discrete event simulations are widely recognized, offering an intuitive grasp for both experienced and inexperienced developers. Given these advantages, the approach adopts the DEVS specification as its foundational framework.

A prevalent theme across the reviewed related works is the paramount importance of scalability, especially when dealing with large-scale IoT simulations. Consequently, one of the primary objectives of this dissertation is to introduce a simulation framework tailored for large-scale IoT scenarios, accompanied by a modeling technique that expedites the creation of extensive use cases.

Rather than opting for the amalgamation of diverse domain-specific simulators into a multi-level simulation, this approach takes a dynamic approach to time advancement and model granularity during simulation. To ensure high scalability in large-scale scenarios, the proposed framework employs varying levels of detail and time advancement for the same simulated entity. These aspects change dynamically during the simulation based on the user's interests. A real-world analogy can be drawn to a magnifying glass, which zooms in with great detail on its focal point while keeping the surroundings constant. This approach enables simulations of expansive scenarios, such as the deployment of a new city-wide service, allowing researchers to simultaneously study relationships across the entire area and delve into intricate processes within individual entities or hardware in the loop.

Similar to the multi-level simulations discussed in Chapter 3, this approach employs multiple levels of detail to simulate complex scenarios. However, instead of implementing different levels using distinct domain-specific simulators, this approach defines simulation hierarchy levels by manipulating time advancement and model detail within the utilized models. While this introduces an upfront modeling cost, as different models with varying levels of detail must be developed, the reusability of these models mitigates this as a one-time expense.

In practice, the switching and substitution of models during simulation runs occur through transitions between low-detail atomic DEVS models and comprehensive network DEVS models. These network models provide a finer-grained understanding of the atomic model's processes and represent a new simulation level. Consequently, a delicate equilibrium between models of specific interest and those providing essential background information effectively reduces the number of generated simulation events. Additionally, by grouping coarse and fine-grained models into partitions, well-established PADS (Parallel and Distributed Simulation) techniques for discrete event simulators can be applied.

To implement these lower-level models in the framework, OMNeT++, complemented by its INET extension, is employed. OMNeT++ serves as the foundation for modeling detailed interactions among ”smart things” and their interplay. This approach leverages OMNeT++’s capabilities to provide a versatile and efficient solution for managing varying levels of detail and time advancement within the simulation, achieving the required scalability for large-scale IoT simulations.

4.5.1. Level Hierarchy and Structure - Expanding DEVS

The proposed approach introduces a novel model type known as *hierarchical atomic* to define the simulation framework within the DEVS specification. This specialized model combines the features of both atomic and network DEVS models and introduces additional functionalities for state transfer and model selection. Mathematically, this new model can be represented as:

$$hatomic \equiv \langle atomic, network, \{transport\}, select\rangle, \tag{4.4}$$

Here, *atomic* and *network* represent the contained atomic and network DEVS models, respectively. {transport} denotes the set of transport functions responsible for transferring state information between models, and *select* is the function responsible for choosing the currently active model.

The detailed definition of the *hierarchical atomic* model is as follows:

$$\begin{aligned} hatomic \equiv \langle & S_A, X_A, X_N, Y_A, Y_N, \\ & D, \{M_N\}, \{I_N\}, \{Z_N\}, \delta_{int}, \delta_{ext}, \delta_{con}, \\ & \lambda, ta, select, \{transport\}\rangle, \end{aligned} \tag{4.5}$$

Let’s break down the components of this definition:

- S_A, X_A, X_N, Y_A, and Y_N represent the state and input/output sets for the atomic (A) and network (N) parts of the model.
- D denotes the set of possible states of the model.
- M_N represents the set of network models within the *hierarchical atomic* model.
- I_N represents the set of input functions for the network models.
- Z_N represents the set of output functions for the network models.
- δ_{int}, δ_{ext}, and δ_{con} are the internal, external, and confluent transition functions.
- λ represents the output function.
- ta is the time advance function.

- *select* is the function responsible for selecting the currently active model.
- *transport* includes the transport functions necessary for transferring states between the atomic and network models, namely

  $$transport_{A \to N} : S_A \to S \subset \cup_i S_i \in D, for S_i \in D_i \tag{4.6}$$

  and

  $$transport_{N \to A} : S \subset \cup_i S_i \in D \to S_A, for S_i \in D_i \tag{4.7}$$

  .

These transport functions are essential for managing the transition between simulation levels. It's important to note that these functions don't need to be isomorphisms since the network model is expected to be more expressive than the atomic model. Therefore, only a subset of possible states from the network model's components is utilized.

In this hierarchical approach, the different contained models define the boundaries of simulation levels. The encapsulated atomic DEVS model resides on the same level as the new *hierarchical atomic* model, while the encapsulated network DEVS model belongs to the level below. This transition between levels is marked by the transport functions.

Notably, models within the same level share similar step sizes in time advancement. If there's a significant discrepancy in the time step sizes among network models belonging to the same level, it can impact the efficiency of the simulation. For optimal performance, it's crucial to ensure that the time advancement characteristics of models within the same level are reasonably consistent.

This novel *hierarchical atomic* model offers a powerful mechanism for managing multi-level simulations efficiently, balancing computational resources, and enabling detailed observations when required. It leverages the strengths of both atomic and network DEVS models while introducing essential features for seamless state transfer and model selection, contributing to the overall effectiveness of the proposed simulation framework.

4.5.2. Model reuse and the model tree

The inherent organizational characteristics of DEVS models, both atomic and network, coupled with the DEVS framework's property of *closure under coupling*, offer a powerful foundation for constructing complex models. This versatility in model composition is a cornerstone of the innovative approach presented in this dissertation. Figure 4.1 visually illustrates the ease with which the newly introduced model, integral to the hierarchical architecture, can function interchangeably as a plain atomic DEVS model within network models at all levels of the hierarchical tree structure. Furthermore, the atomic models nested within the encompassing network model can also be substituted with the newly introduced model. This dynamic model substitution strategy forms the basis for establishing distinct hierarchical levels, each characterized by varying levels of detail and time advancement within the simulation.

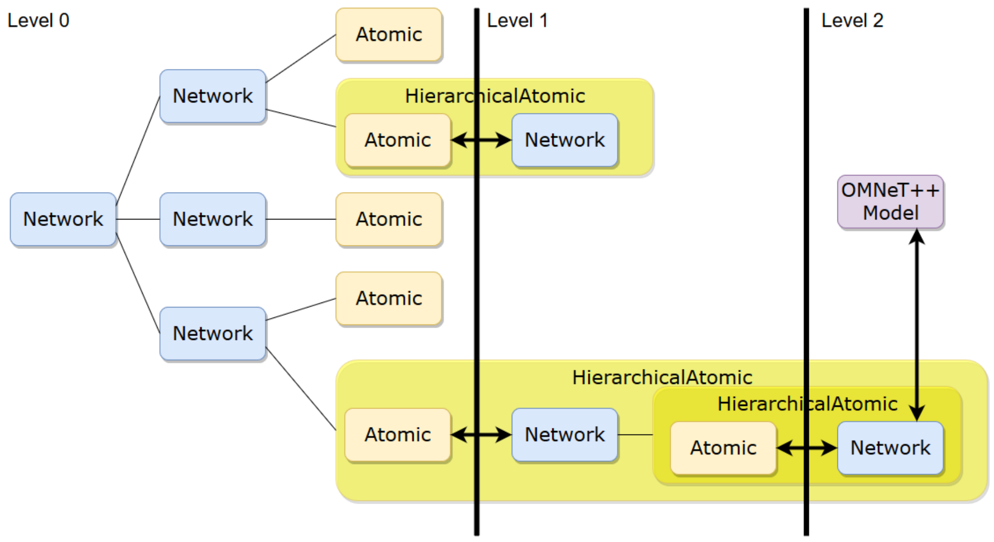


Figure 4.1.: *The model tree and organization of hierarchy levels*

The implications of this tree-like organization of simulation levels extend to the overarching modeling approach and influence the potential for parallelization within the sequential simulation framework.

Firstly, with respect to the modeling approach, it is advisable to adopt a bottom-up methodology. In this approach, the most intricate and detailed level of modeling is initially designed comprehensively and then deconstructed into smaller components, represented as leaf nodes in the hierarchical model tree. The efficiency gained from dynamically exchanging models with varying levels of detail depends on the specific simulation scenario. More efficient model partitions along the tree may only emerge through iterative evaluation during simulation. The bottom-up construction methodology ensures that the core aspects of the scenario are developed only once and can subsequently be partitioned and tailored as needed. Additionally, when considering models that can be interchanged, such as entire protocol stacks or spatially distinct segments of the environment, the potential for extensive model reuse becomes evident. By efficiently leveraging the capability to swap models with more or less detailed representations of the simulated system, complex scenarios, like simulating a user's journey through a smart city while dynamically swapping models to focus on relevant details, become feasible.

In terms of implementation, the strategic placement of integrated domain-specific simulators within the model tree is of paramount importance. This placement can be achieved either by distributing simulators across the nodes of the tree or by adopting a hierarchical structure with one domain-specific simulation kernel per level.

Secondly, in the context of potential partitioning techniques for parallelization, careful consideration must be given to the placement of different networks. Poor partitioning choices within models with a high density of encapsulated networks could negatively impact simulation performance. Therefore, modeling for a parallel scenario may necessitate more thorough planning and consideration compared to other modeling approaches.

During the implementation of the proposed simulator, the decision was made to distribute multiple instances of a simulator across the hierarchical model tree. This strategic placement within the introduced *hierarchical atomics* offered a more flexible implementation of functionality while maintaining a clear separation between the simulator responsible for advancing a specific model and the model itself. Furthermore, as the proposed approach involves the concurrent interchange of models across various levels of the tree, deploying a single instance of a domain-specific simulator per level would not be practical, as it could potentially create bottlenecks that hinder multiple models throughout the simulation.

In summary, the concept of model reuse within a tree-like model hierarchy is a fundamental and ingenious aspect of the proposed simulation approach. It fosters adaptability, scalability, and efficient resource utilization, rendering it a pivotal component of the innovative simulation framework.

4.5.3. Synchronization

When transitioning between an atomic and network model and vice versa, the potential for synchronization errors looms. These errors may arise from previously scheduled autonomous events that could become invalid during the model exchange process. To comprehend the operation mode of the implemented abstract simulator, it's important to note that following an autonomous event, the imminent model is rescheduled using its time advance function. This detail provides a foundation for simplifying synchronization procedures: If model exchanges exclusively occur during events, the subsequent rescheduling of the encompassing model significantly reduces the likelihood of incorrectly scheduled models.

However, depending on the specific implementation of the Future Event Set (FES), internal events may still be scheduled inappropriately, either too early or too late. For instance, during a switch from a detailed, slower-advancing network model to the respective atomic one, there's a possibility that internal events from one of the network's faster-advancing components are still present in the FES. While a straightforward solution involves maintaining a future event schedule that accommodates at most one event per atomic model, the reality is often more complex. Various applications may require different scheduling strategies, and not all of them may align with the requirements of this approach. OMNeT++, for instance, is equipped with several potential scheduling classes, emphasizing the necessity of addressing the synchronization challenge directly.

In light of these considerations, the proposed new model type must autonomously manage its scheduling conflicts and devise responses to erroneously scheduled autonomous events. While models typically track their internal time using their time advance function and the elapsed time since the last internal event, which generally adheres to a correct time advance pattern for the model, this mechanism becomes unreliable when internal event occurrences are in question. Therefore, the model must possess the capability to access global simulation time. This can be achieved in various ways, such as permitting individual models to access the simulation environment or simulation kernel, similar to the approach adopted by OMNeT++. Alternatively, global simulation time can be provided to the model with each function call to one of the state transition functions or the output function.

In essence, addressing synchronization challenges effectively within the proposed approach is essential for maintaining the integrity and accuracy of large-scale IoT simulations, where model exchanges and dynamic adjustments play a crucial role in optimizing performance and resource utilization.

4.6. Implementation of the Simulation Core

In this chapter, we delve into the intricate details of implementing the lightweight simulator, which was previously introduced in section 4.5. Specifically, we explore the seamless integration of this simulator with the OMNeT++ framework, shedding light on the symbiotic relationship between the two simulation environments, as depicted in Figure 4.2.

The primary focus of this chapter is twofold. Firstly, we present the foundational framework responsible for realizing the abstract simulator, as outlined in Subsection 4.3.5. This framework forms the backbone of our simulation architecture and is instrumental in orchestrating the fundamental simulation processes. Secondly, we delve into the pivotal enhancements required to equip our simulation framework with the capacity to model continuous time hybrid systems. The introduction of continuous time modeling is paramount for the accurate representation of various sensor functionalities within the smart city ecosystem. Furthermore, it empowers our simulations to emulate dynamic events occurring within specific time intervals, a capability of paramount importance in scenarios involving motion and movement within the smart city infrastructure.

Moving forward, we detail the intricacies of implementing time hierarchical models. These models introduce a novel dimension of flexibility, enabling the incorporation of different levels of time resolution into our simulations. This flexibility is invaluable for the simulation of complex systems operating under diverse temporal constraints.

In closing, we address the vital topic of integrating our simulator within the OMNeT++ framework. This integration represents a critical juncture in our simulation endeavor, as it ensures the harmonious coexistence and interaction of our lightweight simulator with OMNeT++, thereby harnessing the collective strengths of both environments for the benefit of our smart city simulation framework.

4.6.1. The core simulation framework

This subsection unveils the fundamental architecture of our core simulation framework, an essential component of our lightweight simulator. To achieve

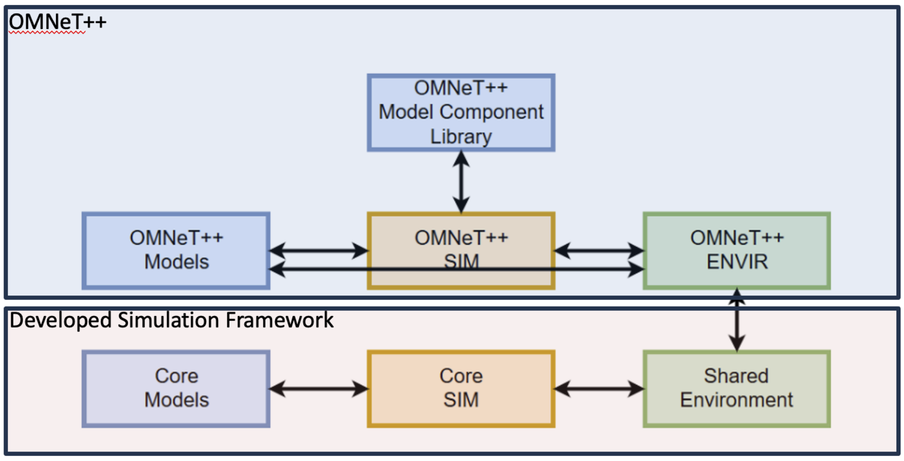


Figure 4.2.: *The architecture of the implementation*

the desired lightweight profile, we carefully considered and adapted the abstract simulator design discussed in Subsection 4.3.5. The conventional hierarchical structure of different processor layers, intertwined with the necessity for continuous control message exchange, appeared to introduce unnecessary overhead. Furthermore, the requirement to manage each model component with its dedicated processor seemed impractical, especially when confronting the vast number of models potentially encountered within a smart city simulation.

Our solution involves flattening the hierarchical logistical structure that envelops the topological hierarchy of DEVS models. The objective is to consolidate all coupled network and atomic models into a single unified level. However, this endeavor goes beyond mere aggregation; it also necessitates the concentration of administrative responsibilities within a single entity. This entity, referred to as the simulator, is designed to assume the pivotal roles of advancing global and local simulation time, orchestrating model state transition functions, and efficiently routing input and output data within the model hierarchy. The result is a versatile simulator capable of supervising an arbitrary number of atomic and coupled models, ensuring the proper execution of simulation procedures.

The simulator functions seamlessly with both atomic and network models, represented as distinct classes. These classes serve as abstract base classes, enabling users of our simulation framework to create models tailored to their specific systems of interest. To simplify the operation on DEVS models, atomic and network models share a common base class.

Interactions within the simulator and between models are managed through events. Events encapsulate critical information, including input and output values, associated models, and the corresponding timestamps. When dealing with future events tied to internal state transitions, the events are stored in a Future Event Schedule (FES), an integral component of the simulator. Beyond its role in managing internal events, the FES also implements the select function for network models (as defined in Section 4.3.3), a critical aspect of the formal DEVS definition. To address situations where multiple models schedule state transitions simultaneously, the FES groups events sharing the same timestamp, thereby ensuring their orderly processing. Additionally, the FES facilitates the implementation of Parallel DEVS (PDEVS) by accommodating delta functions for confluent events.

While our framework suffices for the execution of DEVS simulation models, we extend its functionality to observe and react to state changes, input, and output events. To this end, we introduce the event listener class template, a versatile tool for users to tailor their specific requirements.

Furthermore, we augment the core framework with a lightweight container class that efficiently collects and transfers multiple entities within models and throughout the simulator. To manage the creation and lifetime of these containers, we introduce the object pool, providing an organized and efficient resource management system.

This software system closely resembles the abstract simulator design proposed by Nutaro in [93]. However, notable distinctions from Nutaro's concept are elaborated in subsequent subsections dedicated to specific classes.

The composition of the software system is visualized in Figure 4.3. It comprises instances of the simulator class template, each overseeing numerous DEVS models, both atomic and network, and governing the progression of simulation time. These simulator instances are supported by corresponding Future Event Schedules and Object Pools. Communication between simulators and models is facilitated through instances of events, efficiently routed through the model hierarchy by the simulator and network models. Event listeners stand ready to respond to these events as per the requirements of the user.

In summary, the core simulation framework presented here serves as the foundation for our lightweight simulator. Its adaptability, unified structure, and efficient event management form a robust basis for implementing various DEVS models and accommodating different time and data types, thereby promoting interoperability with other simulation frameworks such as OMNeT++.

Simulator

The Simulator class template (refer to Figure 4.4) forms a vital component of our simulation framework. This class orchestrates the simulation process, and its critical algorithms are elucidated herein. The Simulator operates with an abstract interface, AbstractSimulator, from which it inherits essential methods.

To comprehend its functionality, the primary methods are outlined below:

1. nextEventTime(): This method retrieves the timestamp of the next imminent event.

2. executeNextEvent(): It is responsible for executing the next imminent event.

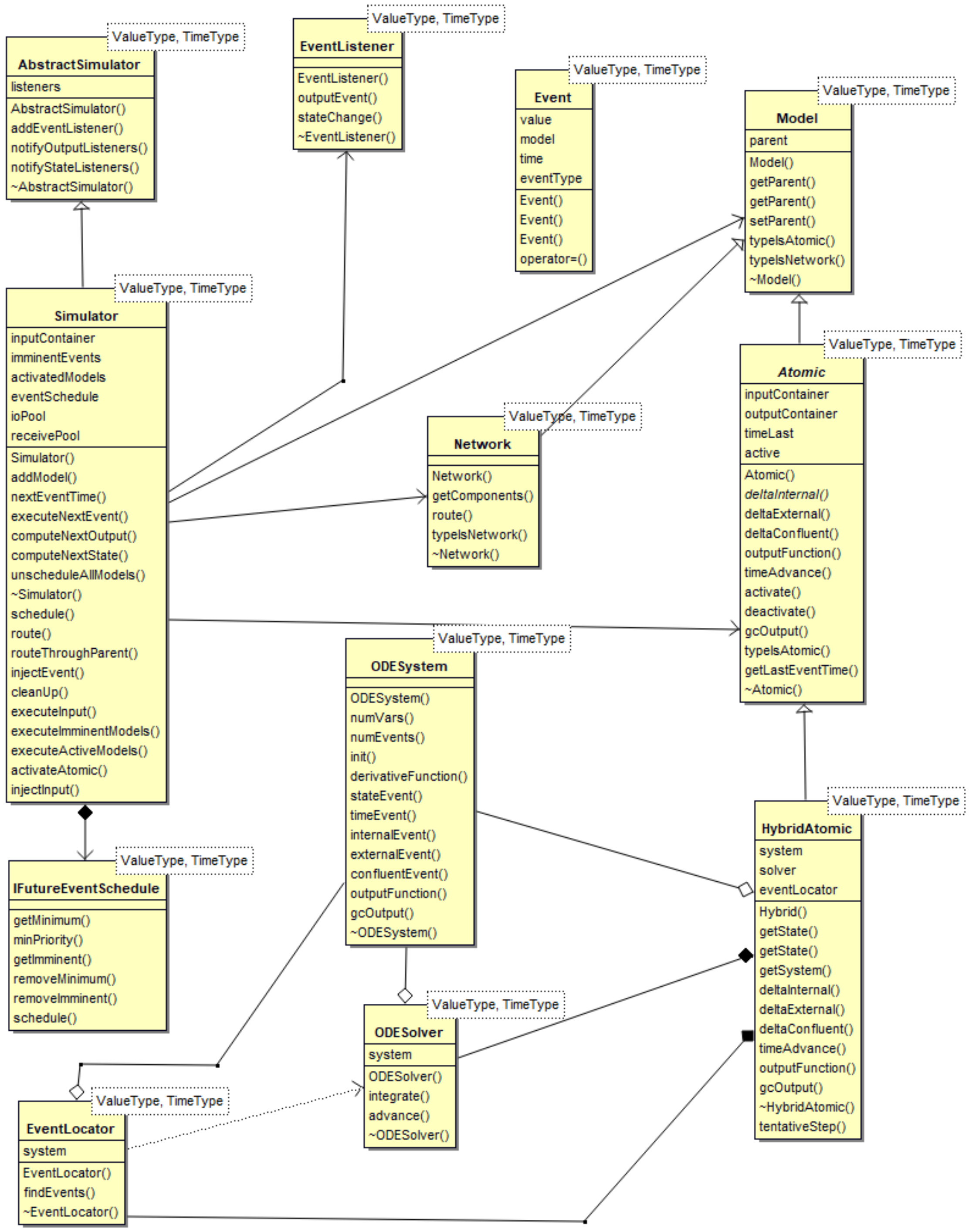


Figure 4.3.: *The core simulation framework as UML*

3. computeNextOutput(): This function calculates the output of imminent atomic models.

4. computeNextState(...): It manages the computation of the next state, considering both internal and external transitions.

5. unscheduleAllModels(): This method unschedules all future events.

The Simulator relies on a Future Event Schedule (FES) to track imminent events. The imminentEvents container holds these events, while the activatedModels container stores models active during the current simulation step. The simulator also employs object pools for efficient object resource management, particularly beneficial for frequently used objects like containers.

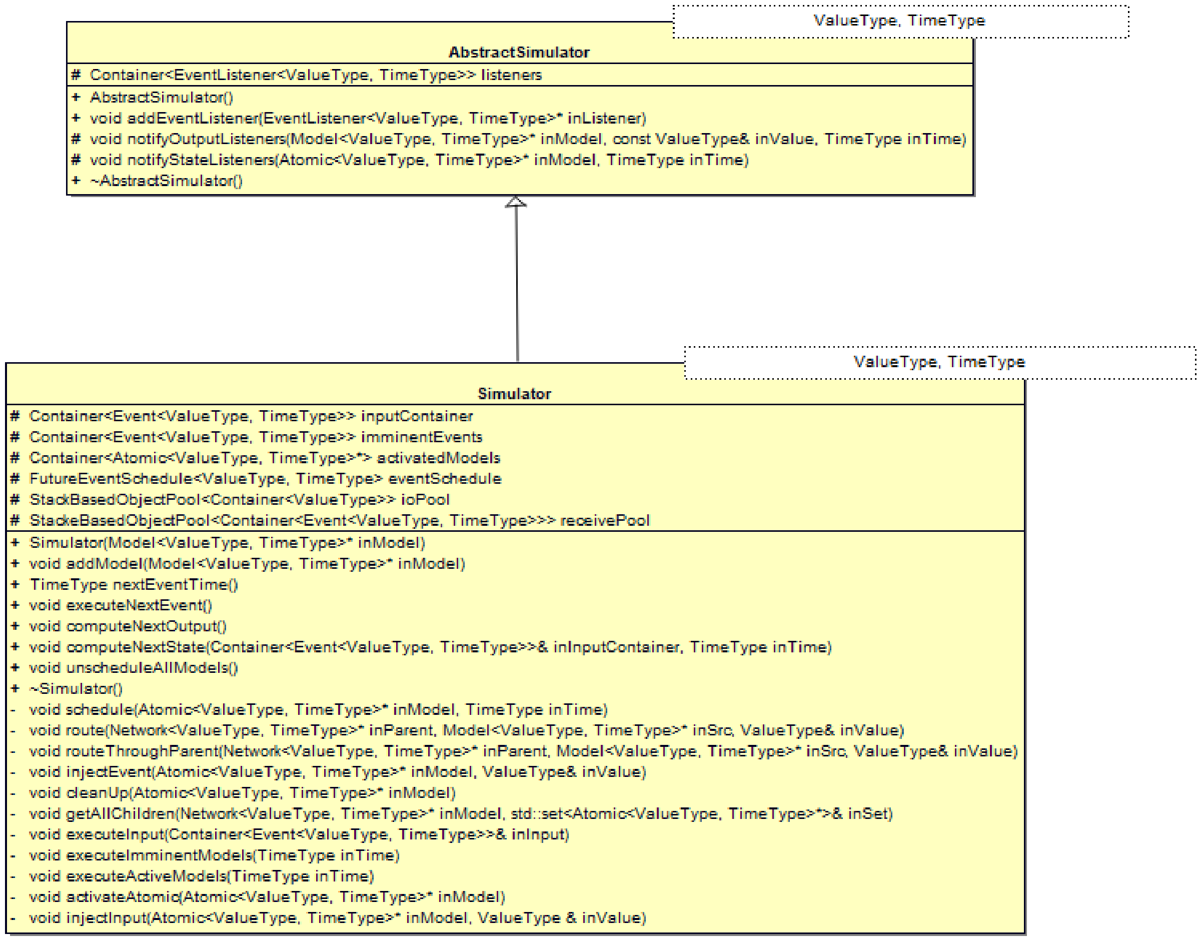

ValueType, TimeType
AbstractSimulator
# Container<EventListener<ValueType, TimeType>> listeners
+ AbstractSimulator()
+ void addEventListener(EventListener<ValueType, TimeType>* inListener)
# void notifyOutputListeners(Model<ValueType, TimeType>* inModel, const ValueType& inValue, TimeType inTime)
# void notifyStateListeners(Atomic<ValueType, TimeType>* inModel, TimeType inTime)
+ ~AbstractSimulator()
ValueType, TimeType
Simulator
# Container<Event<ValueType, TimeType>> inputContainer
# Container<Event<ValueType, TimeType>> imminentEvents
# Container<Atomic<ValueType, TimeType>*> activatedModels
# FutureEventSchedule<ValueType, TimeType> eventSchedule
# StackBasedObjectPool<Container<ValueType>> ioPool
# StackeBasedObjectPool<Container<Event<ValueType, TimeType>>> receivePool
+ Simulator(Model<ValueType, TimeType>* inModel)
+ void addModel(Model<ValueType, TimeType>* inModel)
+ TimeType nextEventTime()
+ void executeNextEvent()
+ void computeNextOutput()
+ void computeNextState(Container<Event<ValueType, TimeType>>& inInputContainer, TimeType inTime)
+ void unscheduleAllModels()
+ ~Simulator()
- void schedule(Atomic<ValueType, TimeType>* inModel, TimeType inTime)
- void route(Network<ValueType, TimeType>* inParent, Model<ValueType, TimeType>* inSrc, ValueType& inValue)
- void routeThroughParent(Network<ValueType, TimeType>* inParent, Model<ValueType, TimeType>* inSrc, ValueType& inValue)
- void injectEvent(Atomic<ValueType, TimeType>* inModel, ValueType& inValue)
- void cleanUp(Atomic<ValueType, TimeType>* inModel)
- void getAllChildren(Network<ValueType, TimeType>* inModel, std::set<Atomic<ValueType, TimeType>*>& inSet)
- void executeInput(Container<Event<ValueType, TimeType>>& inInput)
- void executeImminentModels(TimeType inTime)
- void executeActiveModels(TimeType inTime)
- void activateAtomic(Atomic<ValueType, TimeType>* inModel)
- void injectInput(Atomic<ValueType, TimeType>* inModel, ValueType & inValue)

Figure 4.4.: *The Simulator class template*

The class introduces an interface, IFutureEventSchedule, to ensure flexibility in FES implementations. The interface defines methods such as minPriority(), getImminentEvents(...), removeMinimum(), and schedule(...). By default, our framework uses the FutureEventSchedule, which employs a min heap data structure to manage imminent events efficiently[80]. This heap is organized based on event timestamps, with the root node representing the most imminent event. The FES's main purpose is to determine which event is scheduled next for execution. It optimally handles situations where multiple events are scheduled simultaneously by grouping them and processing them sequentially.

Algorithm 1: The simulation loop

```
1 while nextEventTime() < +∞ do
2     executeNextEvent();
3 end
```

The simulation loop (refer to section 4.3.5) is executed within the Simulator. It starts by determining the timestamp of the next event using nextEvent-Time(). As long as there are future events (nextEventTime() < +∞), the simulation loop iteratively calls executeNextEvent() to execute the events as seen in algorithm 1. The executeNextEvent() function operates in two phases.

In the first phase, it calculates the output of imminent atomic models. For this, it retrieves imminent events from the FES and iterates through them. If an output container does not exist for a model, it creates one and invokes the output function, then routes the output as per the network model's routing information. Models receiving output events are added to the set of activatedModels. The basic algorithm is presented in Algorithm 2.

Algorithm 2: computeNextOutput algorithm

```
1  if outputFunction already executed then
2      return;
3  else
4      imminentEvents ← FES.getImminentEvents();
5      foreach event ∈ imminentEvents do
6          if outputContainer exists then
7              break;
8          else
9              create outputContainer;
10             event.getModel().outputFunction(event);
11             routeOutput();
12         end
13     end
14 end
```

The second phase focuses on computing the new state of active models. It involves internal transitions for imminent models and external transitions for models within activatedModels. The simulator checks for external input in its input container and routes it to associated models. It also executes the internal transition function for imminent models if no external input is associated with them. If external input and autonomous actions coincide, the simulator calls the confluent transition function. Next, activated models transition to their new states based on their external transition functions.

The simulator maintains an input container for external inputs, and this container is cleared after each iteration of the simulation loop. This is used to process external input, routing it to the appropriate models and calculating the external state transitions.

Furthermore, the simulator is responsible for routing input and output events between models. The routing method uses the event source, its value, and the parent model to deliver events. If the destination model is atomic, the input is injected into it, and the model is marked as activated. If the destination is a network model, the simulator recursively routes the events until an atomic model is reached. The routing continues until either an atomic model is encountered or the source model has no parent model (indicating the top-level network model).

Models

In the context of our simulation framework, models refer to both atomic and network DEVS models(refer to section 4.3). These models share common features and an interface (as seen in Figure 4.5) that is fundamental to the functioning of the framework. Here, we will delve into the key aspects of these models.

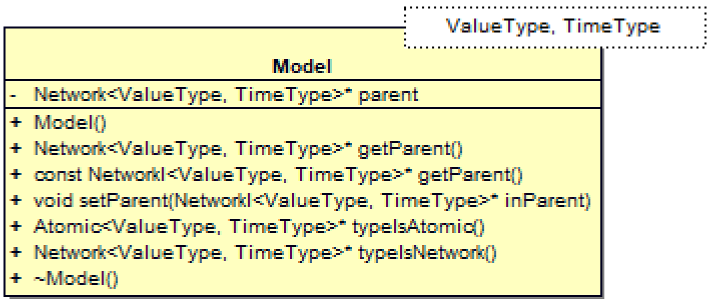

ValueType, TimeType
Model
- Network<ValueType, TimeType>* parent
+ Model()
+ Network<ValueType, TimeType>* getParent()
+ const NetworkI<ValueType, TimeType>* getParent()
+ void setParent(NetworkI<ValueType, TimeType>* inParent)
+ Atomic<ValueType, TimeType>* typeIsAtomic()
+ Network<ValueType, TimeType>* typeIsNetwork()
+ ~Model()

Figure 4.5.: *The model class template*

1. **Hierarchical Structure**: Every model, whether atomic or network, is part of a hierarchical structure. This structure is created by organizing models into different levels of network models, which, in turn, couple atomic models together. To facilitate this hierarchical organization, each model possesses a pointer to its parent network model. This pointer allows a model to access its parent within the hierarchical structure. Models can set their parent using dedicated get and set methods.

2. **Type Identification**: Models need to inform the simulator of their specific type—whether they are atomic or network models. This type identification is crucial because the simulator treats atomic and network models differently concerning scheduling and state transitions. Models achieve this by implementing the isTypeAtomic() and isTypeNetwork() functions. These functions return a null pointer and are expected to be overridden by the respective model classes. By knowing the type of a model, the simulator can appropriately manage its behavior.

Importantly, models in our simulation framework do not need to be aware of the simulator or any other underlying infrastructure of the simulation engine. This separation of concerns ensures modularity and allows models to focus solely on their specific functionality and logic within the simulation.

In essence, models in our framework are the building blocks of the simulation. They can represent various components and subsystems of the simulated system, whether they are atomic entities or network structures. The hierarchical arrangement, parent-child relationships, and type identification mechanisms enable models to interact effectively within the simulation framework, providing a foundation for creating complex simulations.

Atomic Models The implementation of atomic models in our simulation framework adheres to the atomic DEVS formalism and is encapsulated within the Atomic class template. This abstract class template serves as a blueprint for concrete atomic models that are part of a simulation. The interface of the Atomic class template is depicted in Figure 4.6.

Attributes The Atomic class template incorporates four essential attributes for its core functionality:

- timeLast: This attribute stores the simulation time at which the model was last scheduled.
- active: It is a flag that indicates whether the model is currently active and part of the schedule.
- inputContainer and outputContainer: These containers enable the interaction of the model with the external world, allowing the simulator to inject input and collect output.

Access Methods Access to these attributes is facilitated through dedicated get and set methods. Additionally, the deactivate(...) function serves to deactivate the model, returning the input and output containers to the object pool and facilitating garbage collection.

Functions The Atomic class template includes a set of pure virtual functions that must be implemented by concrete models that inherit from it. These functions define the behavior of the atomic model within the simulation:

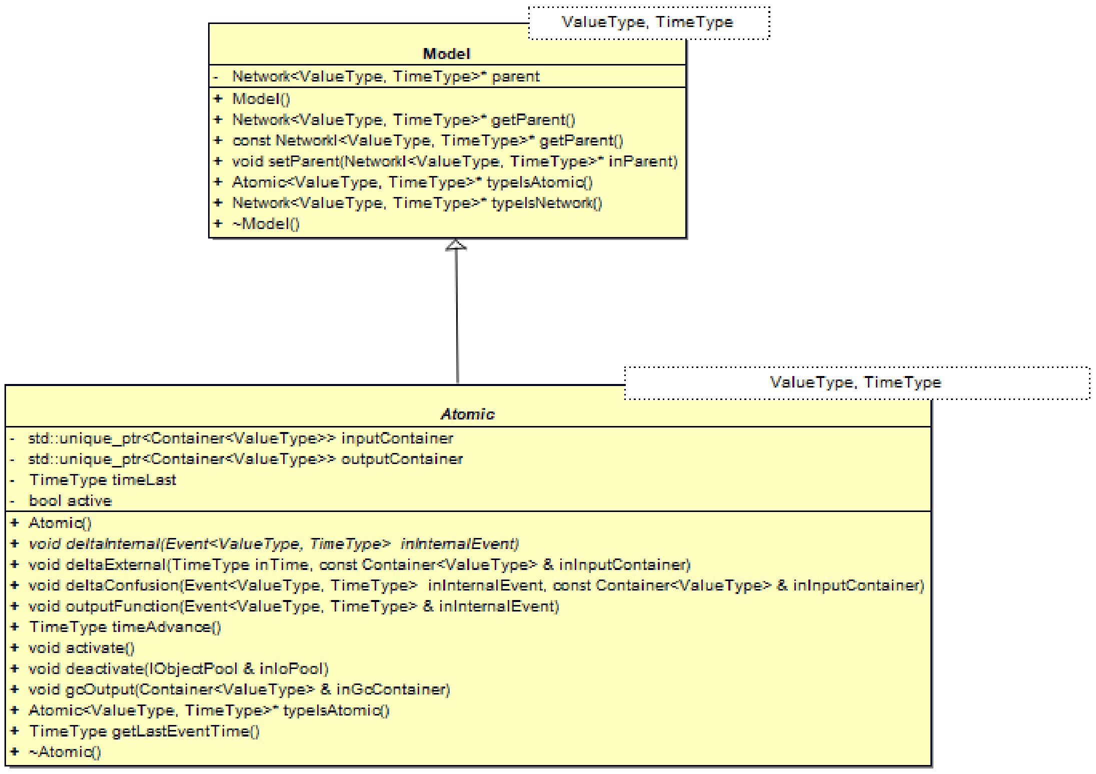

ValueType, TimeType
Model
- Network<ValueType, TimeType>* parent
+ Model()
+ Network<ValueType, TimeType>* getParent()
+ const NetworkI<ValueType, TimeType>* getParent()
+ void setParent(NetworkI<ValueType, TimeType>* inParent)
+ Atomic<ValueType, TimeType>* typeIsAtomic()
+ Network<ValueType, TimeType>* typeIsNetwork()
+ ~Model()
ValueType, TimeType
Atomic
- std::unique_ptr<Container<ValueType>> inputContainer
- std::unique_ptr<Container<ValueType>> outputContainer
- TimeType timeLast
- bool active
+ Atomic()
+ void deltaInternal(Event<ValueType, TimeType> inInternalEvent)
+ void deltaExternal(TimeType inTime, const Container<ValueType> & inInputContainer)
+ void deltaConfusion(Event<ValueType, TimeType> inInternalEvent, const Container<ValueType> & inInputContainer)
+ void outputFunction(Event<ValueType, TimeType> & inInternalEvent)
+ TimeType timeAdvance()
+ void activate()
+ void deactivate(IObjectPool & inIoPool)
+ void gcOutput(Container<ValueType> & inGcContainer)
+ Atomic<ValueType, TimeType>* typeIsAtomic()
+ TimeType getLastEventTime()
+ ~Atomic()

Figure 4.6.: *The atomic class template*

- deltaInternal(...): This function defines the behavior of the model in response to internal events.
- deltaExternal(...): It specifies the behavior of the model in response to external input events.
- deltaConfluent(...): This function handles events that are both internal and external (confluent events).
- outputFunction(...): This function determines the model's output behavior.
- timeAdvance(): It defines the model's time advance function, influencing its scheduling within the simulation.
- gcOutput(...): This function is used for cleanup when the model is no longer in use.
- The deltaInternal(...) and deltaConfluent(...) functions serve as sinks for scheduled internal events from the simulator. This enables atomic models to access the current simulation time, a crucial requirement for their behavior.

- Type Identification: Finally, the class also implements the typeIsAtomic() function, which is inherited from the Model class. This function returns a self-pointer and aids in identifying the model's type as atomic.

In essence, the Atomic class template provides the fundamental structure and behavior that atomic models must adhere to within our simulation framework. Concrete atomic models inherit from this template and implement the necessary functions to define their specific behavior and interactions within the simulation.

Network Models The Network class template plays a pivotal role in our simulation framework, serving as the foundation for modeling complex hierarchical structures of coupled atomic models. Despite its simplicity in terms of attributes, this class template is crucial for facilitating the interactions between atomic models within a network.

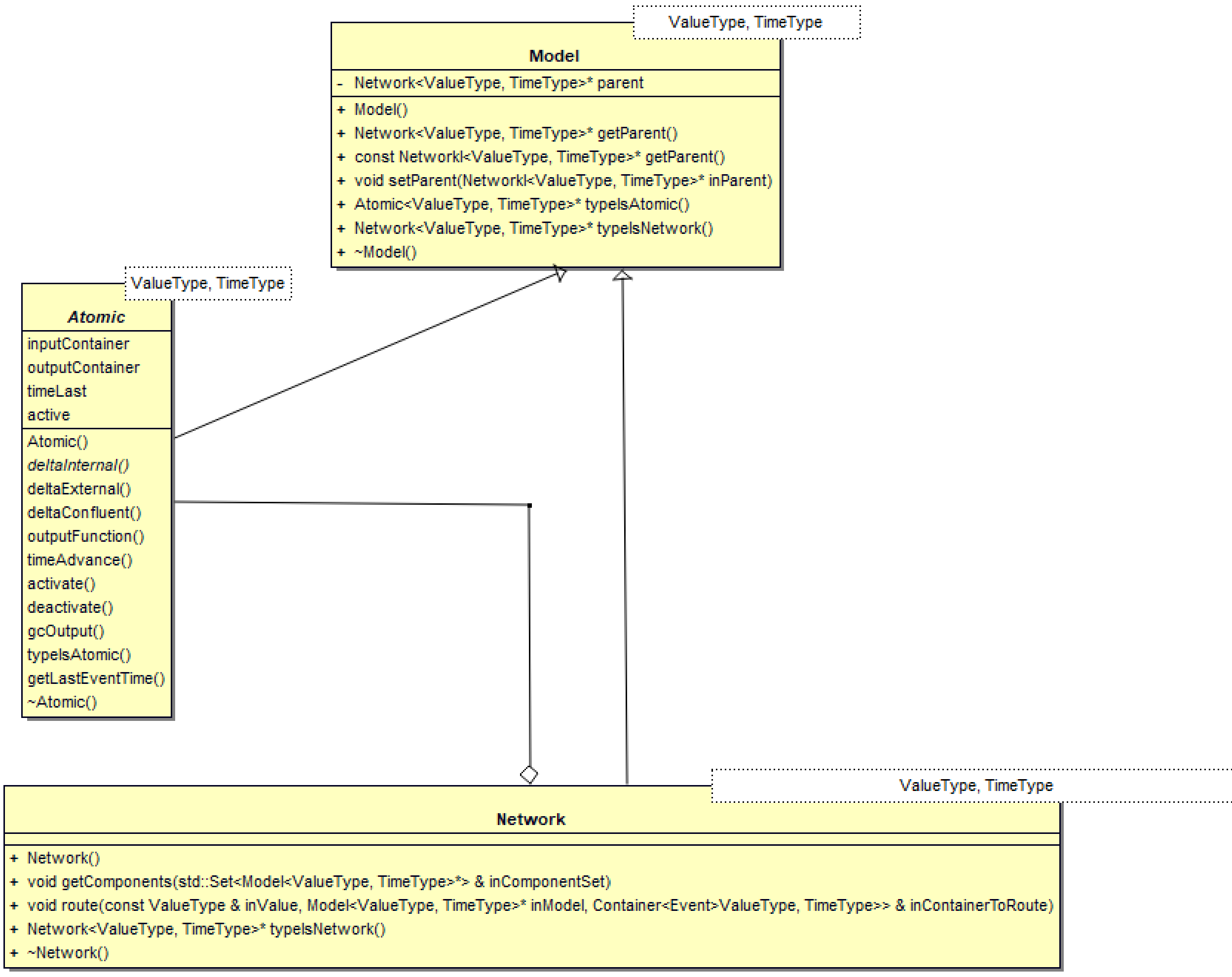

ValueType, TimeType
Model
- Network<ValueType, TimeType>* parent
+ Model()
+ Network<ValueType, TimeType>* getParent()
+ const Networkl<ValueType, TimeType>* getParent()
+ void setParent(Networkl<ValueType, TimeType>* inParent)
+ Atomic<ValueType, TimeType>* typeIsAtomic()
+ Network<ValueType, TimeType>* typeIsNetwork()
+ ~Model()
ValueType, TimeType
Atomic
inputContainer
outputContainer
timeLast
active
Atomic()
deltaInternal()
deltaExternal()
deltaConfluent()
outputFunction()
timeAdvance()
activate()
deactivate()
gcOutput()
typeIsAtomic()
getLastEventTime()
~Atomic()
ValueType, TimeType
Network
+ Network()
+ void getComponents(std::Set<Model<ValueType, TimeType>*> & inComponentSet)
+ void route(const ValueType & inValue, Model<ValueType, TimeType>* inModel, Container<Event>ValueType, TimeType>> & inContainerToRoute)
+ Network<ValueType, TimeType>* typeIsNetwork()
+ ~Network()

Figure 4.7.: *The network class template*

Attributes The Network class template (refer to Figure 4.7) does not include any attributes, as its primary function is to provide a structural framework for connecting atomic models within a network.

Functions

1. getComponents(...) (Pure Virtual): This function is required to be implemented by any concrete network model that inherits from the Network class template. Its purpose is to inform the user about all the atomic models that are coupled together within the network. This function serves as a means for the user to access and manage the atomic models within the network effectively.

2. route() (Pure Virtual): The route() function is an essential component used by the simulator to obtain routing information for both input and output events of the atomic models within the network. It defines how events are routed through the hierarchical structure of the network to their respective destinations. This is crucial for managing the flow of information between atomic models, ensuring proper communication and interaction.

3. typeIsNetwork(): Similar to the typeIsAtomic() function in the Atomic class, the typeIsNetwork() function is implemented in the Network class template. It returns a self-pointer and aids in identifying the model's type as a network model.

In essence, the Network class template is a fundamental building block within our simulation framework, allowing the creation of hierarchical structures of coupled atomic models. It provides the necessary functions for managing and routing events between atomic models, enabling the modeling of complex systems with intricate interactions. Models that inherit from the Network class template must implement the getComponents(...) and route() functions to define the specific structure and behavior of the network.

Hybrid Models

The incorporation of continuous-time hybrid systems into our simulation framework, as defined in subsection 4.3.7, draws inspiration from the work presented in [93]. Here, we provide an overview of the essential methods and components that enable the simulation of hybrid models within our developed simulator.

HybridAtomic Class The HybridAtomic class (refer to Figure 4.8 serves as the core component for simulating hybrid models. It employs three additional classes, namely ODESystem, ODESolver, and EventLocator, to manage and execute the hybrid modeling functionality effectively.

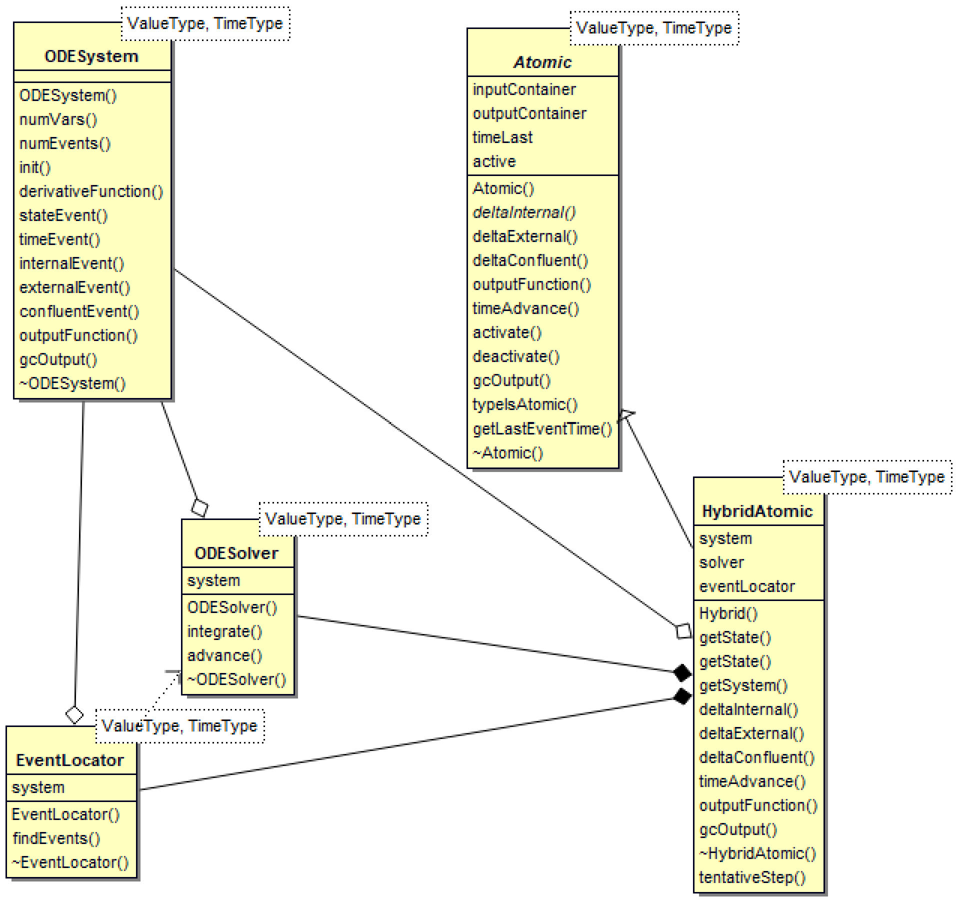


Figure 4.8.: *HybridAtomic*

Attributes

- State Variables: The HybridAtomic class includes attributes to store the state variables of the discrete event model represented by the hybrid system.
- Event Flag: An additional flag within this class plays a pivotal role in indicating the occurrence of an event.

External Classes

- ODESystem: This class encapsulates the continuous-time behavior of the hybrid model, providing the necessary functionalities to manage and solve ordinary differential equations (ODEs).
- ODESolver: Responsible for solving the ODEs associated with the hybrid model, the ODESolver class ensures accurate numerical solutions, allowing for the simulation of continuous-time dynamics.

- EventLocator: The EventLocator class specializes in identifying events within the continuous-time simulation, facilitating the synchronization of discrete and continuous components.

In essence, the HybridAtomic class, along with its associated components, enables the simulation of hybrid models by seamlessly integrating discrete event-driven behavior with continuous-time dynamics. The interaction between these components allows for the accurate representation of systems with both discrete and continuous behaviors, making our simulation framework versatile and capable of modeling a wide range of complex systems.

ODE System The ODESystem serves as a foundational class that must be extended to implement specific hybrid models. It defines several crucial methods:

- stateEventFunction(...): This method computes state event functions, which help identify state transitions within the continuous dynamics of the hybrid model.
- timeEventFunction(...): Similar to the state event function, this method computes time event functions, which capture specific time points where events occur.
- derivativeFunction(...): The derivative function calculates the rate of change of the continuous state variables. This function plays a fundamental role in advancing the continuous dynamics of the hybrid system.

Additionally, the ODESystem class includes methods for implementing discrete behaviors:

- internalEvent(...): This method handles internal events, which occur within the continuous-time component of the model.
- externalEvent(...): External events are managed by this method, allowing the model to react to discrete events from its environment.
- confluentEvent(...): In cases where internal and external events coincide, the confluent event function is used to manage their interactions.
- outputFunction(...): The output function computes the model's outputs based on its current state.

By providing these methods, the ODESystem class forms the basis for implementing the continuous and discrete behaviors of hybrid models.

ODE Solver The ODESolver class is responsible for advancing the continuous state variables of the hybrid model over time. It defines two key methods:

- Integrate(. . .): This method advances the system's state from one time point to another, accommodating the continuous dynamics. It returns the actual integration step size used during the process and updates the states stored within the HybridAtomic class.

- advance(. . .): Similar to Integrate, this method also advances the continuous state but guarantees advancement up to a specific time point.

By utilizing these methods, the ODESolver class ensures the accurate progression of the continuous-time component of the hybrid model.

Event Locator The EventLocator class focuses on identifying state events within a fixed interval. It is an abstract class, and its implementations must define the findEvents(. . .) method. This method is crucial for detecting state events by analyzing the continuous state of the hybrid model and the end state at the conclusion of the fixed interval. It relies on a specified integration method to locate state events accurately. When a state event is found, the EventLocator class records the time of its occurrence, the model's state at that moment, and a flag indicating which threshold function triggered the event.

In summary, the ODESystem, ODESolver, and EventLocator components work together to enable the accurate simulation of hybrid models by seamlessly integrating continuous and discrete behaviors. These components provide the necessary tools and methods for modeling complex systems with both continuous and discrete dynamics.

4.6.2. Hierarchy Middleware

In this section, we embark on the practical realization of the novel model type introduced in Chapter 4.5, known as the Hierarchy Middleware. This intricate component forms a crucial part of our simulation framework, facilitating the modeling of complex hierarchical structures within the simulated environment.

To comprehensively understand the architecture and functionality of this Hierarchy Middleware, we will first delve into the intricate web of inheritance relationships and associations that underlie its design. This discussion serves as a foundational exploration, shedding light on the structural underpinnings of our model, paving the way for a deeper analysis of its operational intricacies.

Subsequently, we will transition to a more granular examination of the implementation, delving into specific sections of the codebase that carry particular significance. While prior discussions have primarily focused on abstract algorithms and high-level concepts, this part of the section aims to provide a practical, code-level perspective on how the Hierarchy Middleware operates within our simulation framework. Through this lens, we can gain valuable insights into the technical underpinnings that empower our model to handle the complexities of hierarchical systems in the context of discrete-event simulation.

Hierarchical Atomic

The Hierarchical Atomic model, embodied in the HierarchicalAtomic class template, constitutes a pivotal component of our simulation approach, encapsulating both atomic and network models. This amalgamation, depicted in Figure 4.9, facilitates the seamless integration of atomic and network model behaviors within a single framework.

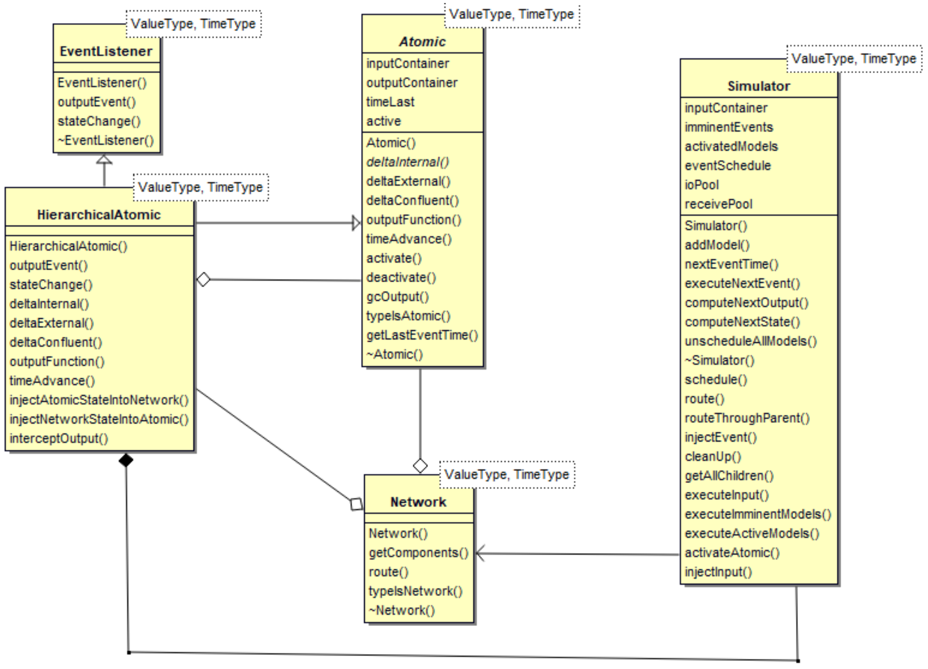


Figure 4.9.: *Hierarchical Atomic*

The HierarchicalAtomic class template is designed to function as a wrapper around an arbitrary model. It can encapsulate both an Atomic model and a Network model, functioning as an intermediary between them. From the perspective of an external Simulator (as discussed in subsection 4.6.1), the HierarchicalAtomic behaves like a regular atomic model. However, it orchestrates the switch between its internal encapsulated models, ensuring that their behaviors are faithfully represented and synchronized.

A key distinction is that the HierarchicalAtomic itself does not verify whether the behavior of the encapsulated Network model provides a more detailed specification of the Atomic model's behavior. This responsibility falls upon the user, who must choose the appropriate models to encapsulate.

This class is instrumental in the management of its encapsulated models' execution and state synchronization. It maintains a private flag, hierarchyActive, to determine which model is currently active. This flag is toggled through the method requestHierarchySwitch, which ensures synchronization between the models and avoids instantaneous switches, allowing the encapsulated models to react to events before switching.

The HierarchicalAtomic class inherits from EventListener, which enables it to register itself as an event listener in the encapsulated Simulator. When the internal Simulator triggers events in the HierarchicalAtomic, it can check these events against its list of output models to determine which outputs are relevant to the current model.

The synchronization mechanism (refer to Figure 4.10) is managed through the synchronizationError flag, which detects scheduling errors arising from finer-grained time advances within the encapsulated Network. This flag ensures that events are scheduled correctly, even when switching back and forth between the encapsulated Atomic and Network models.

```
template <typename ValueType, typename TimeType>
void deltaInternal(Event<ValueType, TimeType> inInternalEvent){
  if (!hierarchyActive){
    if (hierarchySwitchRequested){
      injectAtomicstatusIntoNetwork();
      //Schedule Models
      simulator->addModel(this->network, inInternalEvent.time);
      simulator->computeNextState(inputBag, inInternalEvent.time);
      hierarchySwitchRequested = false;
      hierarchyActive = true;
    }else{
      //Detect synchronization errors
      if (inInternalEvent.time < this->timeLast + atomic->timeAdvance()){
        correctedTimeAdvance = this->timeLast + atomic->timeAdvance() -
          inInternalEvent.time;
        synchronizationError = true;
        return;
      }
      atomic->deltaInternal(inInternalEvent);
    }
  }else{
    if (hierarchySwitchRequested){
      simulator->executeNextEvent();
      injectNetworkStatusIntoAtomic();
      simulator->unscheduleAllModels();
      hierarchySwitchRequested = false;
      hierarchyActive = false;
    }else{
      internTimeLast = simulator->nextEventTime();
      simulator->executeNextEvent();
    }
  }
}
```

Figure 4.10.: *HierarchicalAtomic's deltaInternal method*

The timeAdvance method in the HierarchicalAtomic class, as shown in Figure 4.11, calculates the simulation time advance for both encapsulated models. If the HierarchicalAtomic is currently in the Atomic mode and a synchronization error is detected, it corrects the time advance accordingly. When the

Network is active, it scales down the time advances to reflect the finer-grained time resolution. This ensures that scheduled events are executed at the correct times, preventing synchronization errors.

```
template <typename ValueType, typename TimeType>
TimeType HierarchicalAtomic<ValueType, TimeType>::timeAdvance(){
  if (!hierarchyActive){
  if (synchronizationError){
    synchronizationError = false;
    return atomic->timeAdvance() - correctedTimeAdvance;
  }else{
    return atomic->timeAdvance();
    }
  }else{
    if (simulator->nextEventTime() < pdevsInf<TimeType>())
    return simulator->nextEventTime() - internTimeLast;
  else
    return pdevsInf<TimeType>();
  }
}
```

Figure 4.11.: *HierarchicalAtomic's timeAdvance method*

In summary, the HierarchicalAtomic model serves as a crucial bridge between atomic and network models within our simulation framework. It seamlessly switches between these models, ensuring their behaviors are faithfully represented and synchronized, all while preventing synchronization errors. The encapsulation of models allows for a more versatile and modular approach to simulation modeling, where complex hierarchical structures can be built with ease.

4.6.3. OMNeT++ - Integration

OMNeT++ Integration into our existing simulation framework is a pivotal step that empowers our system with the capabilities of OMNeT++, a renowned network simulation tool. This integration enhances the versatility and applicability of our framework, enabling the simulation of complex systems encompassing both discrete event-driven and network-centric behaviors. In the subsequent sections, we delve into the details of this integration.

The integration process requires modifications to OMNeT++'s environment class, enabling seamless interaction with our simulator. This environment class, often referred to as the Simulation Host Environment (SHE), serves as the bridge between OMNeT++ and our simulation framework. In the next subsection on the shared environment, we scrutinize these necessary adaptations, elucidating how they facilitate the coexistence of OMNeT++ and our framework.

Subsequently, in the section on AbstractModuleCoupling, we explore the intricacies of data exchange between the various types of models in both simulators. This includes atomic and network models in our framework, as well as modules and connections in OMNeT++. This data interchange mechanism is vital for achieving a harmonious integration where information flows seamlessly between the two simulation environments.

Shared Environment

To seamlessly merge our core simulation framework, detailed in Section 4.6.1, with OMNeT++, a unified simulation loop known as the Shared Environment is essential. This shared environment acts as a unifying interface, providing users access to the functionalities of both simulators while presenting critical information regarding simulation management and noteworthy events.

OMNeT++ offers three pre-built environments: a basic command-line environment, a *Tk* environment, and a *Qt* environment, each equipped with its simulation loop. However, to facilitate the integration of our simulation framework with OMNeT++, users have the flexibility to create their custom environments, which can then be registered and selected at runtime. For this purpose, OMNeT++ offers the *cRunnableEnvir* class as an interface to implement custom environments.

In this integration effort, we opt to utilize the *EnvirBase* class as a foundational component. This class is employed by OMNeT++ for its pre-built environments, offering a range of built-in functionalities, such as setup and cleanup procedures before and after simulation runs. The *EnvirBase* class will serve as the base for our shared environment, enhancing the efficiency of the environment development process.

To harmonize the functionalities of both simulators within the shared environment, we establish a combination of interfaces. The primary interface, ISharedEnvironment, defines the requisite methods necessary to extend any environment built on top of *EnvirBase* into a shared environment. This interface closely collaborates with a data structure that embodies the Model Tree, a concept detailed in subsection 4.5.2. The public interface of this data structure is defined by IModelTree. Further elaboration on these interfaces will be provided in the subsequent sections.

In an endeavor to streamline the development of the shared environment, we draw upon the existing functionality of OMNeT++'s command-line environment, transferring it to a new class template that also implements the newly introduced interfaces. This approach significantly simplifies the process of creating a shared environment, enabling users to harness the capabilities of both simulators within a unified simulation context.

ISharedEnvironment The ISharedEnvironment interface plays a pivotal role in enabling the seamless integration of our dual simulator environment. Rather than hardcoding the necessary changes directly into the command-line environment class, this interface offers a more versatile approach. It empowers classes that rely on an instance of the shared environment, such as the AbstractModuleCoupling, to function independently of the specific environment implementation. Moreover, it paves the way for deploying more advanced environments in the future. The interface definition is presented in Figure 4.12.

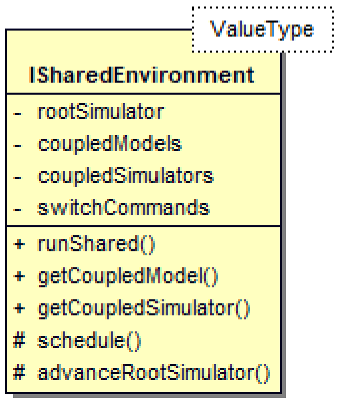


Figure 4.12.: *ISharedEnvironment*

In the context of ISharedEnvironment, the TimeType parameter for all utilized class templates is mapped to OMNeT++'s simtime_t, while the ValueType parameter remains open until the simulation's data type is determined. The interface mandates the implementation of the following methods:

- runShared(): This method encapsulates the core simulation logic required to run both simulators within a single simulation loop. A concrete implementation of this method will be provided later in this section.
- getCoupledModel(. . .) and getCoupledSimulator(. . .): These methods play a pivotal role during the simulation's initialization phase, linking OMNeT++ modules and Atomic models.
- advanceRootSim(): This method is responsible for advancing the simulation managed by the rootSimulator. It is intended to be invoked by SyncEvents, which are injected into OMNeT++'s simulation loop.
- schedule(. . .): To gain access to the protected schedule method within the simulators, the simulator class template needs to befriend the interface. The interface then utilizes the protected schedule method to schedule a model within the simulator's Future Event Set (FES).

Additionally, the interface includes a set of utility functions for convenient access to specific models or simulators from the model tree.

Furthermore, ISharedEnvironment maintains a data structure containing pointers to HierarchicalAtomics and timestamps at which they should transition between their encapsulated models. This data structure is accessible to all potential environment subclasses that implement the interface. This shared data structure ensures consistent and synchronized behavior across the shared environment, facilitating the seamless interaction between the two simulators.

IModelTree The core of any potential implementation of ISharedEnvironment hinges upon the underlying data structure used to represent the hierarchy of HierarchicalAtomics and the overall model topology. Although the name might suggest a conventional tree structure, for reasons of efficiency, it's not a strict requirement. The interface can be observed in Figure 4.13, defining the following methods:

- addModel(...): This method, as the name implies, is responsible for incorporating a given model into the data structure. It should also store information about any encapsulated simulators if they inherit from HierarchicalAtomic. This stored information becomes essential for the overloaded method getPathToRootSim, which returns all simulators on a bottom-up path from any given simulator or model.
- getPathToRootSim(...): This method plays a critical role in navigating the hierarchy. Given a simulator or model, it returns all simulators encountered along the path to the root simulator.

The data structure represented by IModelTree serves as the backbone for efficient and effective interaction between models and simulators in our dual environment. It allows for precise traversal of the model hierarchy, ensuring that the interactions between models and simulators are well-coordinated and synchronized. This is especially crucial in the context of the shared environment, where both simulators must operate in harmony to produce accurate and meaningful results.

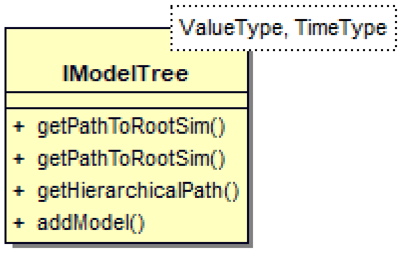


Figure 4.13.: *IModelTree*

SharedSequentialScheduler and SyncEvent Introducing a new class of events, this section outlines an essential mechanism for synchronizing the rootSimulator and OMNeT++. This class extends OMNeT++'s existing cEvent class and thus requires the implementation of the pure virtual method, execute. This method's role is straightforward; it casts OMNeT's active environment to the ISharedEnvironment interface and then invokes its advanceRootSim method.

These specialized events, referred to as SyncEvents, are generated by instances of the SharedSequentialScheduler class template. This class template expands upon OMNeT's cSequentialScheduler and introduces the critical functionality needed to harmonize both simulators within a unified simulation loop. When the scheduler is queried for the next imminent event, it conducts a meticulous examination of both simulators' Future Event Sets (FES). If it determines that the next event of the root Simulator should take precedence over the earliest event in OMNeT++, it creates and returns a new SyncEvent. This event triggers synchronization, ensuring that both simulators operate seamlessly together in the same simulation loop. This tight synchronization is crucial to maintain consistency and accuracy throughout the simulation.

SharedEnvironment With all the necessary interfaces and classes in place, it's now possible to construct a shared environment. Essentially, this environment should implement the previously introduced interfaces and make appropriate adjustments to OMNeT++'s simulation method.

The simulateShared method, showcased in Figure 4.14, is a customized version of OMNeT++'s CmdEnv's simulate method. To keep the focus on the modifications, unmodified code is encapsulated within the helper functions setupSimulateShared and endSimulateShared.

```
template <typename ValueType>
void SharedEnvironment<ValueType>::simulateShared(){
  Speedometer speedometer = setupSimulateShared();
  try{
    if (!opt->expressMode)
      runSimulateShared();
    else
      runSimulateSharedExpressMode(&speedometer);
  }catch (cTerminationException& e){
    if (opt->expressMode)
      doStatusUpdate(speedometer);
    loggingEnabled = true;
    stopClock();
    deinstallSignalHandler();
    stoppedWithTerminationException(e);
    displayException(e);
    return;
  }catch (std::exception& e){
    if (opt->expressMode)
        doStatusUpdate(speedometer);
    loggingEnabled = true;
    stopClock();
    deinstallSignalHandler();
    throw;
  }
  endSimulateShared(speedometer);
}
```

Figure 4.14.: *SharedEnvironment's simulateShared method*

setupSimulateShared configures various parameters required for the simulation loop and the collection of statistics and status information. Subsequently, it calls the runSimulateShared (or runSimulateSharedExpressMode) function, which handles the actual simulation.

Since most administrative tasks are already managed by ISharedEnvironment and SharedSequentialScheduler, only minor adjustments are needed. These adjustments are made within the runSimulateShared and runSimulateSharedExpressMode functions. Figure 4.15 illustrates the runSimulateShared function, and runSimulateSharedExpressMode functions are analogous.

The primary change is within the core simulation loop, where it's checked if the next event of the rootSimulator occurs before the next event of OMNeT++. This is achieved using the getNextSimTime method, which retrieves the arrival time of the first event in OMNeT++'s Future Event Set (FES) and then restores the event to its original state. If any events in the rootSimulator's schedule are scheduled to occur before the next OMNeT++ event, the rootSimulator advances its simulation until an OMNeT++ event occurs. After the OMNeT++ event is executed, the loop repeats, processing any reactions to that event.

The logic ensures that the simulators' internal state transitions are perfectly synchronized, contributing to the overall consistency and accuracy of the simulation.

AbstractModuleCoupling

In order to seamlessly connect atomic models with OMNeT++'s cSimpleModules for use as transportation networks, a key component is introduced: the abstract class template AbstractModuleCoupling (refer to Figure 4.16). This class inherits from OMNeT++'s cSimpleModule and the EventListener class template. The TimeType parameter of EventListener is set to simtime_t in OMNeT++, as this is the shared simulation's time unit. However, the ValueType parameter remains open and must be defined for specific use cases, as it depends on the data type used in the simulation.

The main function of this class is to facilitate the translation of inputs between the core simulation framework described in section 4.6.1 and OMNeT++. To inject messages into OMNeT++ models, the class implements EventListener's outputEvent method (refer to Figure 4.17). When an output event occurs in the core simulation, this method is invoked. It subsequently triggers the translateOutput and prepareAndSend methods, both of which are pure virtual methods that must be implemented by concrete modules inheriting from AbstractModuleCoupling. Additionally, the setContext calls are vital. They ensure that the output is correctly associated with the respective module and prevents issues where messages would be owned by OMNeT's cDefaultList instead of the module itself.

On the other hand, to inject OMNeT++ messages into the core simulation framework, the handleMessage method of cSimpleModule is employed. This process is depicted in Figure 4.18.

In essence, when a message arrives in an OMNeT++ module, it's translated into the corresponding data type for the simulation. This translated message

```
template <typename ValueType>
void runSimulateShared(){
  while (true){
    if (!this->switchCommands.empty()){
      auto switchCommand = this->switchCommands.front();
      if (switchCommand.time <= getSimulation()->guessNextSimtime()){
        for (auto hatomicIter = switchCommand.models.begin();
          hatomicIter != switchCommand.models.end(); hatomicIter++)
          (*hatomicIter)->requestHierarchySwitch();
        this->switchCommands.erase(this->switchCommands.begin());
      }
    }
      cEvent *event = getSimulation()->takeNextEvent();
      if (!event)
        throw cTerminationException("Scheduler interrupted while waiting");

        // flush *between* printing event banner and event processing, so that
        // if event processing crashes, it can be seen which event it was
        if (opt->autoflush)
          out.flush();

        // execute event
        getSimulation()->executeEvent(event);

        // flush so that output from different modules don't get mixed
        cLogProxy::flushLastLine();

        checkTimeLimits();
        if (sigintReceived)
          throw cTerminationException("SIGINT or SIGTERM received, exiting");
  }
}
```

Figure 4.15.: *SharedEnvironment's runSimulateShared method*

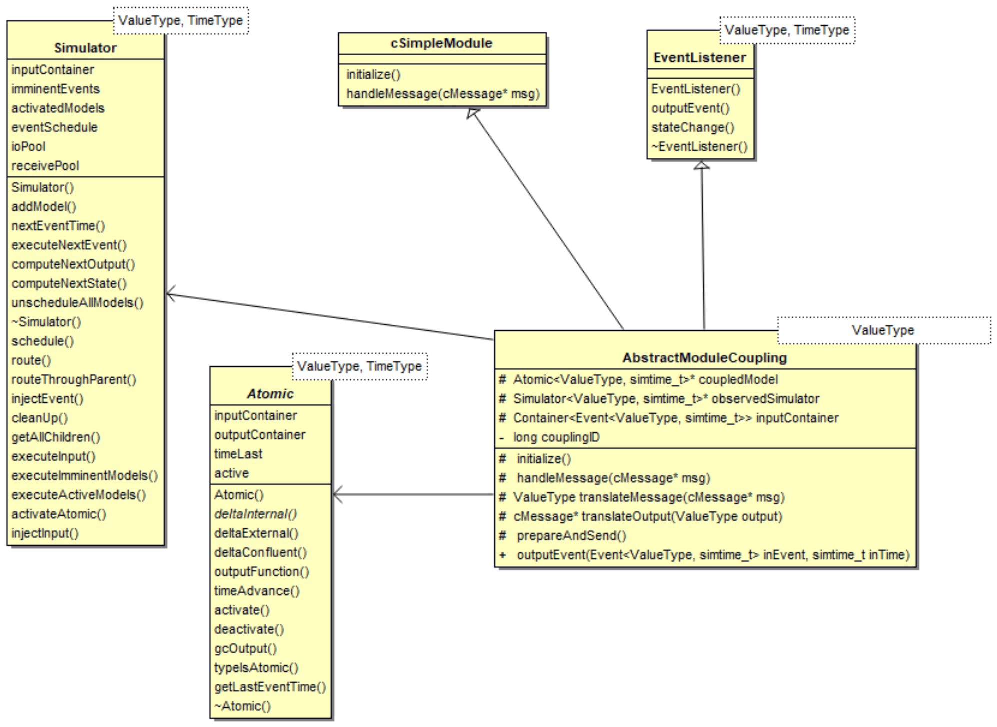


Figure 4.16.: *AbstractModuleCoupling*

is then encapsulated into an event and injected into the simulator managing the model. The simulator's computeNextState method is called to execute the input, triggering an external state change. Finally, the environment overseeing the simulation is notified. This step is essential because different simulator entities managing the simulation aren't aware of each other, so the environment must traverse the hierarchy of wrapped simulators from the receiving model to its root and reschedule the hierarchical models that contain the receiving one.

In concrete implementations of ModuleCoupling, methods such as translateOutput, prepareAndSend, and translateMessage must be implemented. These methods rely on pointers to coupledModel and observedSimulator, which are obtained during OMNeT++'s initialization phase. The couplingID parameter is read from the respective NED files during initialization, used to request the necessary pointers from the active environment, and register itself as an event listener for the coupled model on the observed simulator.

4.7. Hardware in the loop

In many instances, the need arises for interactions with objects external to the simulation environment, such as the authentic VICINITY network. These interactions may encompass direct engagement with real-world objects or merely exerting control over entities within the simulation. In this context, VICINITY

```
template <typename ValueType>
void outputEvent(pdevs::Event<ValueType, simtime_t> inValue, simtime_t inTime){
  if(inValue.model == coupledModel){
    getSimulation()->setContext(this);
    cMessage* msg = new cMessage((messageName).c_str());
    msg = translateOutput(inValue.value, msg);
    prepareAndSend(msg);
    delete msg;
    getSimulation()->setGlobalContext();
  }
}
```

Figure 4.17.: *AbstractModuleCoupling's outputEvent method*

```
template <typename ValueType>
void handleMessage(cMessage *msg){
  ValueType translatedMsg = translateMessage(msg);
  Event<ValueType, simtime_t> inputEvent(coupledModel, translatedMsg, getSimulation()
    ->getSimTime());
  inputBag.insert(inputEvent);
  if (observedSimulator){
    observedSimulator->computeNextState(inputBag, getSimulation()->getSimTime());
    auto env = check_and_cast<ISharedEnvironment<ValueType>*>(getSimulation()
      ->getActiveEnvir());
    env->omnetInputArrived(observedSimulator, coupledModel);
    inputBag.clear();
  }else{
    throw new PDEVSCouplingError();
  }
  delete msg;
}
```

Figure 4.18.: *AbstractModuleCoupling's handleMessage method*

emerges as an invaluable communication platform due to its device-agnostic and standards-agnostic attributes. To facilitate the seamless integration of a specific infrastructure into VICINITY, the pivotal component required is an Adapter that furnishes the necessary REST API for communication with the VICINITY Agent.

In the context of infrastructures simulated via our simulation framework, establishing a means of communication with the VICINITY Agent beyond the simulation environment becomes imperative. To address this requirement, we have implemented a socket-based approach for interacting with the VICINITY Agent. This approach obviates the need for an entire network simulation, offering a streamlined mechanism to communicate with the VICINITY Agent, thereby enhancing the efficiency and flexibility of interactions between the simulated environment and the VICINITY network.

4.7.1. OMNeT++ socket server

To establish seamless communication between the OMNeT++ simulation and the VICINITY Agent, the integration of a socket server becomes imperative. The primary function of this server is to facilitate the reception of API calls from the VICINITY Agent, mediate the exchange of pertinent data with the OMNeT++ simulation, and subsequently dispatch a response.

The fundamental challenge in this endeavor lies in the synchronization of socket requests with the discrete event scheduler inherent to OMNeT++. Initially, the simulation must be executed in real-time, a feat achieved by harmonizing event-time with wall-clock time. This entails the deliberate delay of event execution within the scheduler. The temporal gap between two consecutive events serves as an opportune window for the scheduler to listen to and process socket data. When necessary, new events can be injected into the event queue to handle received data.

An alternative approach is adopted wherein a dedicated socket server-thread is employed to preprocess all inbound and outbound socket data. This methodology serves to minimize the computational overhead incurred by the simulation, as only relevant data is exchanged with the simulation thread. The server-thread effectively manages other processing tasks, including communication and protocol-related overhead, without imposing any noticeable impact on the simulation. Consequently, the simulation scheduler awaits notifications from the server-thread rather than socket requests. This synchronization is achieved through the utilization of promise and future objects, integral components of the C++ standard library.

Figure 4.19 provides a visual representation of the core synchronization concept between the socket server-thread and the simulation thread, elucidating the orchestration of communication within the hardware-in-the-loop functionality.

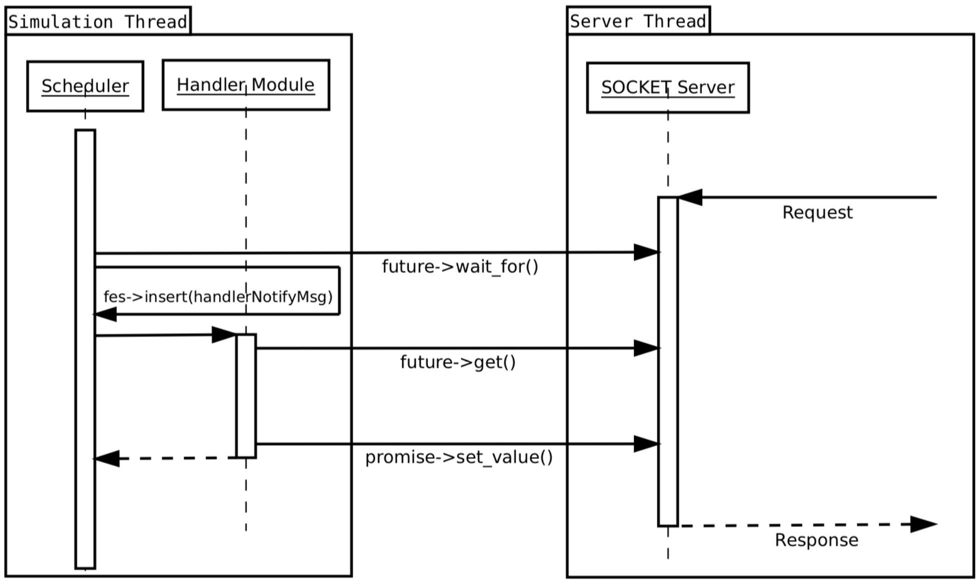


Figure 4.19.: *Simulation- and Server-Thread synchronization*

4.7.2. OMNeT++ VICINITY Adapter

To enable seamless communication between VICINITY and simulated objects, regardless of the network employed within the simulation, a simulated access-point plays a pivotal role. This access-point effectively operates as an intermediary, functioning as a router to facilitate the exchange of requests and responses between VICINITY and the simulation, which includes relaying and protocol conversion as required.

The access-point module, denoted as VicinityAccesspoint, collaborates closely with VicinityAdapter, a subclass of SocketServer. The latter is responsible for managing the socket server thread, which handles incoming requests from VICINITY. In the context of synchronization, as illustrated in Figure 4.19, VicinityAdapter assumes the role of the socket server, while VicinityAccesspoint serves as the handling module. Below, we delve into the functionality of these two pivotal modules:

VicinityAdapter Upon initialization, VicinityAdapter is configured with a designated port for listening. Upon invoking startAdapter(), the method returns a future-object. As soon as a request arrives, it triggers the availability of an object of type RequestStruct. This object encapsulates all pertinent request data. Additionally, a promise-object is provided, which is subsequently utilized to establish a ResponseStruct. Within this structure, all response-related data is conveyed, along with a fresh ‘promise‘-object designed for handling the next incoming request.

VicinityAccesspoint The VicinityAccesspoint module assumes the pivotal role of routing adapter requests into the simulation. Furthermore, it is tasked with furnishing object-discovery data upon request by the adapter. To compile all requisite data for object discovery, the access-point initiates a broadcast of messages of type ObjectDiscoveryPkt (refer to Figure 4.20). These broadcasts include *isRequest=true* flags, addressing all connected modules. Modules that should be accessible via VICINITY subsequently respond with their own ObjectDiscoveryPkt message, marked as *isRequest=false*, while including their respective object-data in JSON format within the objectJson field.

The access-point then consolidates a JSON object encompassing all the received ObjectDiscoveryPkt responses. This JSON object is served upon request by the adapter. When an incoming request pertains to setting or reading a property or action, the access-point forwards this request to the relevant module, employing a VnetPkt message. The recipient module processes the request and issues a response back to the access-point, again through a VnetPkt message. Upon processing the message, the access-point generates a response destined for the adapter, thus completing the request. In cases where a targeted module fails to respond within a specified time frame, a timeout event occurs, prompting the access-point to issue a *VININITY_RESPONSE_NOT_FOUND* response to the adapter, thus concluding the request with an error code.

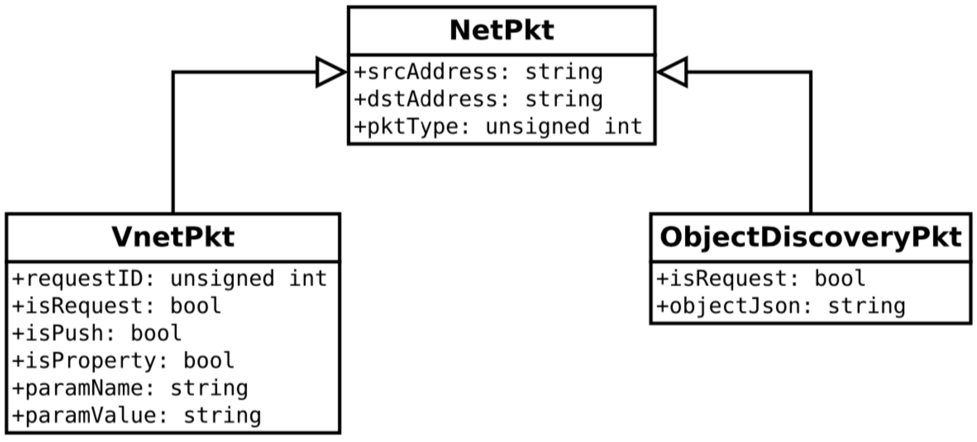


Figure 4.20.: *Class diagram of messages used inside the simulation*

4.7.3. Simulation Scheduler

As elucidated in section 4.7.1, the OMNeT++ scheduler necessitates adaptation to seamlessly integrate the adapter. The original SocketRTScheduler was purpose-built for actively listening on a single socket during the time gaps between simulation events. In our framework, however, the management of sockets is delegated to a dedicated thread. This arrangement notably simplifies the role of the scheduler, as it only needs to monitor a single future-object and, upon any changes to this object, notify the handler module, in this case, the access-point.

Upon initiating the simulation, the access-point module (VicinityAccesspoint) registers a notification message and a pointer, which consistently points to the most recent future-object from the adapter-thread, with the scheduler. Given that the scheduler operates as a real-time scheduler, it must wait until the next scheduled event time (handled in cSocketRTScheduler::takeNextEvent()). During this waiting period, the scheduler remains idle, patiently anticipating the readiness of the future-object.

When the designated future-object finally becomes ready, the simulation time is updated to align with the actual time, effectively accounting for the time spent blocked on the future-object. Subsequently, the notification-message assigned to VicinityAccesspoint is scheduled for immediate delivery at the current time. Upon receiving this notification-message, the access-point is primed to accept and process data from the future-object. Furthermore, the access-point takes on the responsibility of updating the future-object, to which the scheduler holds a reference. This ensures that the scheduler remains prepared for processing subsequent requests. In cases where the future-object is deemed invalid, the scheduler simply enters a sleep state, awaiting either a timeout or the arrival of the next scheduled event.

4.8. Human in the loop

To address the stipulated requirements elucidated in Section 4.2.1, we shall embark on a journey to devise a functional specification that serves as the bedrock for our conceptual framework.

Expanding upon the insights gleaned in the subsection on the Mobility Pilot Site, we shall pivot our focus towards the realm of the smart parking use-case. This particular scenario hinges on the intricate interplay between events and the sensors embedded within individual parking lots, i.e., the context arises when a vehicle seeks parking.

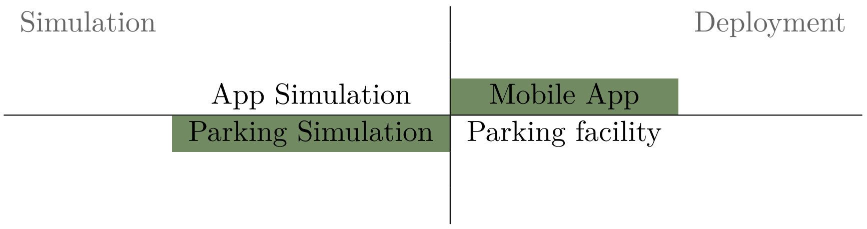


Figure 4.21.: *Simulation and Deployment Matrix: At the vertical the environments (simulation and deployment) are mirrored. The horizontal states the related use-cases of each environment.*

It is essential to bear in mind the nuances illustrated in Figure 4.21, which factor into the comprehensive simulation of parking scenarios. This encompasses a holistic consideration of the parking ecosystem. Furthermore, we must remain cognizant of the need to adhere to practical constraints while still delivering a solution commensurate with the allocated timeframe.

In light of these requisites, the core mission at hand revolves around the establishment of a seamless conduit of communication between the various entities involved. Additionally, a validation of the efficacy of this communication framework is imperative, manifesting as a compelling proof-of-concept for the underlying technology stack.

Mobility Pilot Site Tromsø

Smart parking	**Parking with smart sensor integration**
Priority parking	Specific parking space request is prioritised
Emergency parking	Automatic allocation for first responder
Shared parking	Reimbursed for sharing parking space

Table 4.1.: *The use-cases of the Mobility Pilot Site Tromsø with their descriptions*

The crux of our Human-in-the-Loop demonstration is centered predominantly on the mobility pilot site situated in Tromsø. This choice has been made deliberately, with a particular emphasis on the intricacies and challenges intrinsic to this locale. At its core, the primary use-case scenario within this chosen pilot site is poised to tackle a common urban conundrum - the dearth of real-time information regarding available parking spaces. The consequences of this deficiency are well-documented, with a substantial portion of urban traffic congestion arising from the prolonged quest for unoccupied parking slots. It's worth noting that the smart parking pilot site is seamlessly integrated into a larger context, specifically an assisted living and healthcare facility. Within this context, an underground garage facility takes center stage, providing a total of 32 parking spaces. This facility encompasses two crucial features: two Electric Vehicle (EV) charging stations and four designated handicap parking spaces. Enumerated in Table 4.1 are the envisaged use-cases tailored to the Tromsø facility. However, the focal point of this use case part orbits squarely around the smart parking use-case, leveraging this distinct scenario as a testing ground. To facilitate the seamless operation of the underground garage, each parking space is equipped with a suite of technical apparatus. This ensemble comprises a camera sensor, entrusted with the responsibility of vehicle authentication; a visual indicator, employed for signaling parking space availability; and a ground sensor, instrumental in detecting the presence of a vehicle within a given parking slot. The schematic representation of this concept is succinctly depicted in Figure 4.22, encapsulating the essential elements of this smart parking infrastructure.

Mobile App

When delving into mobile app development, a pivotal decision revolves around the choice of the mobile operating system, predominantly between Android and iOS. It's noteworthy that as of 2017, these two behemoths jointly dominated a staggering 99.8% of the mobile OS market share [94].

However, for the scope of this dissertation, iOS has been exclusively chosen as the target platform. This decision was steered by several factors, chief among them being the seamless alignment with the ecosystem at Rheinland-

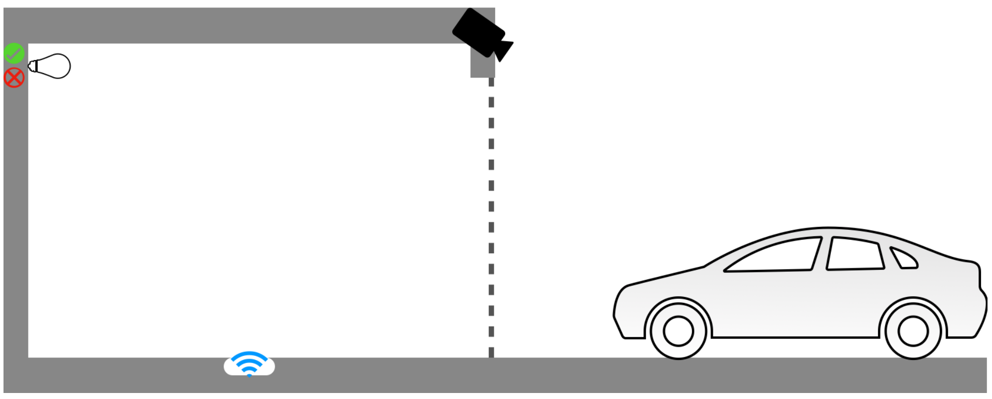


Figure 4.22.: *Conceptional drawing of the functionalities of one parking lot within the Mobility Pilot Site*

Pfälzische Technische Universität Kaiserslautern-Landau (RPTU), particularly within the Cyber-Physical Systems (CPS) group.

RPTU boasts an Apple-friendly environment that is well-integrated with iOS app development tools and resources. This alignment not only streamlines the development process but also ensures a more efficient and effective integration with the broader infrastructure within RPTU, facilitating a more cohesive and harmonized approach..

Cross-Platform towards Native App Development

Even when concentrating primarily on the iOS platform, the issue of whether to design the program as a native or cross-platform application is crucial. This choice specifies the method of development and the tools that will be used during the software lifecycle.

An app is considered to be native if it closely follows the framework and instructions given by the iOS platform. It utilizes iOS-specific functionalities and design principles and is created particularly for iOS devices. A cross-platform program, in comparison, aims for a wider range of compatibility and uses a higher degree of abstraction to access the features offered by both the iOS and Android platforms. A different programming language or framework that is independent of any particular platform is frequently used to implement this abstraction.

The main draw of the cross-platform strategy is its ability to share a single codebase across many platforms, such as iOS and Android, without the need to produce separate versions of the app in other programming languages. However, this convenience has a unique set of factors to take into account, making the decision between native and cross-platform development complex.

Table 4.2 provides a thorough analysis of the advantages and disadvantages of both strategies to aid in this decision-making process. The evaluation criteria for this project include:

1. **Hardware Support:** This criterion pertains to the ability to access and

leverage hardware features like cameras, GPS, accelerometers, NFC, and more.

2. **Energy Efficiency:** In the mobile realm, minimizing power consumption is of paramount importance. Striving for a battery life of at least a day necessitates careful consideration of every feature's impact on energy consumption.

3. **Performance:** Balancing energy efficiency and performance is a fundamental challenge. The aim is to deliver a stable, high-performing app while maintaining an energy-efficient behavior.

4. **Cost:** The financial aspect is crucial for organizations, especially when considering that native development typically requires separate codebases for iOS and Android, potentially necessitating the hiring of additional developers.

5. **User Experience (UX):** UX is a crucial determinant of an app's success. Graphical elements and aspect ratio compatibility with various modern smartphones significantly impact user comfort and familiarity.

6. **Recyclable Code:** The ability to reuse code across different devices is an important factor for developers, affecting development efficiency and maintenance.

7. **Device Compatibility:** As mobile operating systems evolve, app compatibility is essential. Apps must adapt to new OS versions to remain supported and usable.

Table 4.2 illustrates the fundamental differences between cross-platform and native app development, highlighting the trade-offs between the two approaches. These differences stem from the inherent nature of cross-platform development as a ”third-party” approach, resulting in potential lag in keeping pace with platform updates and advancements.

For the human in the loop interface as part of this simulation framework, the native approach has been chosen. While hardware support, energy efficiency, and performance may be secondary considerations in the initial development phase to achieve a minimum viable product, other factors hold greater weight.

From the perspective of the VICINITY project, cost and code recyclability are not of paramount importance, as the project's primary concern is achieving a top-tier user experience to encourage user engagement. Furthermore, the desire to utilize the latest and most advanced features of the iOS platform aligns with the project's commitment to cutting-edge technologies. Therefore, a native iOS app is the most fitting choice for this endeavor.

App Functionality Enabling seamless communication between an app and a simulation tool necessitates the creation of a user interface that is both

	Native	Cross-Platform
Hardware-Support	+	-
Efficency	+	-
Performance	+	-
Cost	-	+
User-Experience	+	-
Recyclable Code	-	+
Device Compatibility	-	+

Table 4.2.: *Pros and cons of Cross-Platform or Native development*

intuitive and purposeful. A well-designed interface must strike a delicate balance between providing adequate information and features while avoiding overwhelming the user.

Initially, the app should offer users a clear and straightforward entry point, ensuring that the interface does not inundate them with excessive data or features. Providing users with a concise overview of the app's core functionality is essential. This introductory section serves to familiarize users, particularly first-time users, with the app's purpose and capabilities, establishing a sense of comfort and ease of use.

The primary function of the app revolves around smart parking, specifically assisting users in determining the availability of parking spaces. It's important to note that the app operates exclusively within a simulated environment, where users have input control instead of direct interaction with physical vehicles. Consequently, the app will focus on visualizing parking lot occupancy and status, enabling users to interact with the simulation effortlessly.

However, it's worth emphasizing that the graphical user interface of the app will not incorporate additional simulation data or logs. Such data would essentially replicate the functionality provided by the underlying OMNeT++ platform, leading to unnecessary redundancy and potential interface clutter. Thus, the decision to omit these elements in the app's interface serves to maintain a streamlined and efficient user experience.

Lastly, the app's functionality is complemented by an imprint and contact form, facilitating user feedback and communication. This professional touch ensures that users have a channel through which they can provide input, report issues, or seek assistance, contributing to a robust and user-centric application.

In summary, the app's functionality centers on providing users with clear and actionable information related to smart parking within the simulation environment. It maintains a user-friendly and uncluttered interface, focusing on its core purpose while facilitating user feedback and interaction through an integrated contact mechanism.

Accessibility Accessibility must be carefully considered before moving on to the layout design. A crucial concern is making sure the app's user interface is simple to use on a number of devices. Common smartphone sizes, which normally range from 4.7 inches to 6.5 inches, can be used to evaluate this. Table

4.3 offers a grid for categorizing display positions to assist in this assessment.

With Table 4.3 as a reference, the accessibility of the app should be assessed when holding the smartphone in the left hand. Users should be able to navigate the interface by iteratively tapping on the screen using their thumb. It's important to note that the smartphone can also be held in the right hand, in which case the numbering of Table 4.3 should be mirrored. Regardless of hand orientation, different levels of effort are required to reach specific on-screen elements. To quantify these efforts, a metric is introduced:

1. **Easy:** The thumb effortlessly reaches all interface elements, requiring no additional muscle strain or hand movements.
2. **Moderate:** Here, the thumb may need to be stretched or supported by auxiliary hand movements to access certain interface elements.
3. **Hard:** These interface elements demand specific thumb movements or even necessitate the support of the second hand to access effectively. In essence, they are practically unreachable using only the thumb.

13	14	15
10	11	12
7	8	9
4	5	6
1	2	3

Table 4.3.: *Classification of a smartphone screen with a grid of numbers*

To enhance usability, each grid array in Table 4.3 is color-coded according to its assigned accessibility category: green for "easy," orange for "moderate," and red for "hard."

Ultimately, these accessibility metrics inform design decisions, particularly regarding layout and placement of interface elements. Figure 4.23 provides an overlay with an iPhone, offering a visual representation of these considerations and guiding subsequent design choices.

Wireframe The wireframe is an essential early-stage component in the development of any app-driven product. It serves the crucial purpose of providing a clear visualization of the application's appearance and functionality. Wireframes aim to showcase how the application will be used, offering a skeletal framework devoid of any design elements that might distract from its core functionalities. This is achieved by eschewing the use of colors, logos, fonts, or other graphical elements.

In this work, a minimalist approach is adopted, employing plain grey boxes to indicate areas where text could be placed. Boxes with a white cross symbolize areas for images, while boxes featuring a play button icon signify actions to be executed.

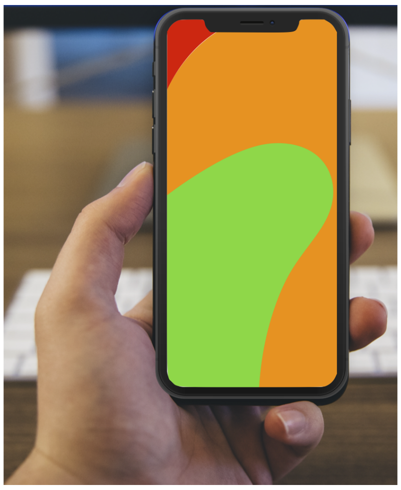


Figure 4.23.: *Accessibility of a smartphone screen with only using one hand*

As depicted in Figure 4.24, the wireframe is derived from the functionalities outlined in Section 4.8 and positioned within the layout of an iOS-capable device. The process involves making certain assumptions about the use of images and the arrangement of elements.

The initial screen of the app corresponds to the general information segment of the functionalities. Here, the application might display essential information such as the detection of the simulation or deployment environment. Subsequently, the user is presented with the choice of either initiating the parking simulation or accessing detailed information via the imprint area.

The wireframe for the middle screen in Figure 4.24 represents the primary functionality of communicating with the parking simulation. To emphasize the need for user action, this screen is intentionally kept minimal, featuring only the essential components: an action button and an indicator displaying the availability status of parking spaces.

Another user option is to navigate to the detailed information screen, which provides explanations and clarifications. Notably, the application includes navigation capabilities, allowing users to return to previous screens or switch between the three main screens, a topic further elaborated upon in Section 5.5.3.

Parking Simulation

Conceptually, the parking simulation may be broken down into three main parts: the camera, the floor sensor, and a visual indicator (see Figure 4.22). Together, these elements carry out a thorough simulation of parking space occupancy.

From an abstract standpoint, it is not difficult to understand how these parts work together. The floor sensor locates a vehicle on the parking surface while the camera takes a picture of the parking area. On the other hand, the visual indicator offers a visual signal to show whether the parking place is

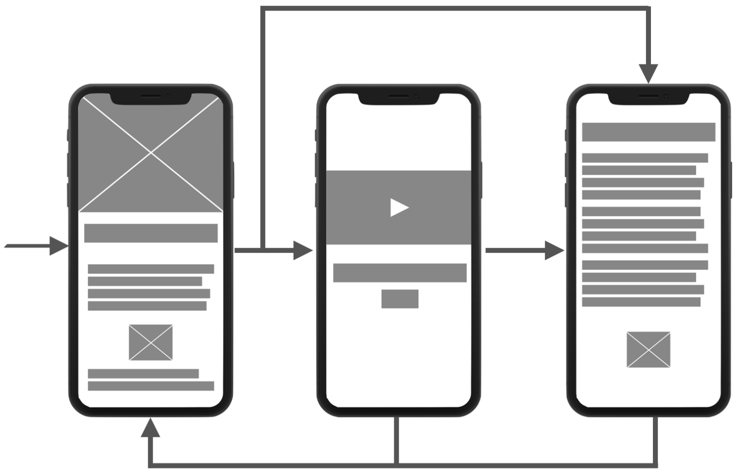


Figure 4.24.: *Wireframe with interaction sequences*

open or occupied.

However, when delving into the details of the simulation's implementation, certain assumptions and simplifications must be considered. To prototype and validate the proof of concept, it is assumed that the camera component readily accepts the presence of any vehicle without prior verification. In practical terms, this means that upon the execution of a command within the app (triggered by the user's action), the camera will consistently interpret this as a valid vehicle arrival.

Both the visual indicator and the floor sensor follow a straightforward operational model akin to a simple switch mechanism. When a command is issued through the app, these sensors respond by switching their state from "parking space free" to "parking space reserved," thereby simulating the occupation of the parking space.

Regarding communication within the simulation environment, the foundational principles established in the OMNeT++ adapter discussed in Section 4.7.2 can be adopted. This means that the core focus of communication within the simulation context will center on the visual indicator and the floor sensor, orchestrating the seamless execution of the parking space occupancy simulation.

Network Connection

To facilitate interaction between the iOS app and the smart parking simulation, a well-defined route must be established. As indicated in Figure 2.1, this involves identifying and connecting the various nodes along this communication path. Consequently, this section aims to consolidate the individual concepts previously described into a coherent network route.

Initiating from the iOS mobile app, which functions as one endpoint, akin to a client in network terminology (as elucidated in Section 2.5.2), the first objective is to establish a connection with the virtual machine housing the smart parking simulation. The OMNeT++ adapter presented in Section 4.7.2, plays a pivotal role in this connectivity. Within the OMNeT++ adapter, two nodes can be discerned: one serving as the gateway to the external network, and the other designated for interaction with the simulation itself, as depicted in Figure 4.19.

The final participant in this intricate network configuration is the smart parking sensor simulation. Thus, the smart parking simulation is interconnected with the OMNeT++ adapter. In summary, a user's request must traverse a total of five nodes in this network architecture before circling back to deliver a response.

The ensuing section will provide a visual representation of this network connection, considering the comprehensive concepts governing each constituent element of this communication framework.

Architectural Model

Having distilled the concepts outlined in previous sections, an architectural model can be synthesized and visually represented, as depicted in Figure 4.25. Each constituent entity is represented, complete with its respective connection nodes indicated by green dots. Additionally, the communication route is delineated by the green line. Consequently, the architecture is designed to ensure that communication adheres to the precisely defined route.

Figure 4.25 underscores the principal focus of this work: the comprehensive functionalities of the iOS app and the smart parking simulation. In this architectural framework, the virtual machine and the OMNeT++ adapter serve as a foundational infrastructure, acting as a facilitating framework. Therefore, they are not illustrated in exhaustive detail but rather as pivotal components of the network architecture.

This architectural model provides a holistic perspective on the network connection and interaction among the various entities involved, thereby offering a comprehensive overview of the system's design and communication pathways.

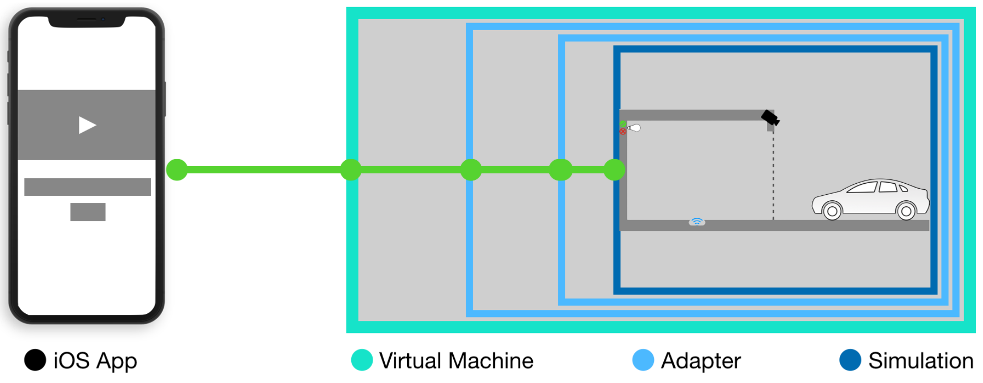


Figure 4.25.: *Holistic view of connection nodes and the endpoints of the route which are represented by the iOS app and parking simulation.*

Chapter 5

Case Study: Smart Energy Use Case

Contents

This chapter embarks on a comprehensive exploration of a case study focusing on the intricacies of a Smart Energy Use Case. Please note that certain portions of the content and findings presented in this chapter have been previously disseminated in conferences such as [86, 87, 88, 89, 90], and have been thoughtfully incorporated into this dissertation to provide a comprehensive and coherent perspective on the subject matter.This chapter commences with an insightful introduction, setting the stage for an in-depth analysis of the case study. It delves into the scenario underpinning the case study, complete with a SysML Model that elucidates its intricacies. Furthermore, the chapter delves into the establishment of a Knowledge Base within SysMD, providing a holistic overview of its structure. Within this realm, it explores the Smart-Power Knowledge Base, scenarios embedded in SysMD, and the multifaceted dimensions of Smart Parking and Smart Home applications. The chapter then transitions into the practical domain, detailing the meticulous implementation of the performance evaluation. It elucidates the smart home network and its module connections, security-, health-, electronics-simulations, and the intricate calculations related to energy consumption. A significant facet of this chapter is the implementation of the Human-in-the-loop interface, which allows for real-time interactions within the simulation. It thoroughly explicates the setup of the OMNeT++ Adapter, Parking Simulation, and the iOS App interface. The chapter culminates in a comprehensive evaluation of the case study's various dimensions. This includes an analysis of SysMD Models, a rigorous assessment of performance, an exploration of Software in the Loop Evaluation, and a profound understanding of Human in the Loop Evaluation. This chapter serves as a holistic exploration of the Smart Energy Use Case within the broader context of IoT network simulation.

5.1. Introduction to the Case Study

The objective of this case study is the development of a sophisticated smart parking and smart home scenario for IoT simulation. This endeavor leverages the simulation approach delineated in Chapter 4.5, thereby harnessing its intrinsic capacity for facilitating large-scale, multi-level simulations. This feature proves indispensable given the complexity of the envisaged use case, entailing the simulation of numerous households, cars and power generators, each replete with an array of interconnected devices. One notable advantage of the chosen simulation approach is its seamless integration with OMNeT++ [95], a powerful and versatile simulation tool. OMNeT++, fortified with its INET framework, furnishes a conducive environment for simulating network traffic, a pivotal facet in any IoT scenario. Additionally, OMNeT++ boasts an integrated power management tool, a salient feature germane to our use case. The utility of this embedded power management tool lies in its ability to perform comprehensive power consumption assessments for individual consumers (households, cars, etc.). Through meticulous simulations, it enables the accurate prediction of a house's power consumption profile. This predictive capability is instrumental in optimizing energy production and allocation, ensuring that energy generation aligns precisely with anticipated consumption patterns. By adhering to this principle, energy surplus is minimized, leading to enhanced energy efficiency and sustainability in smart home environments.

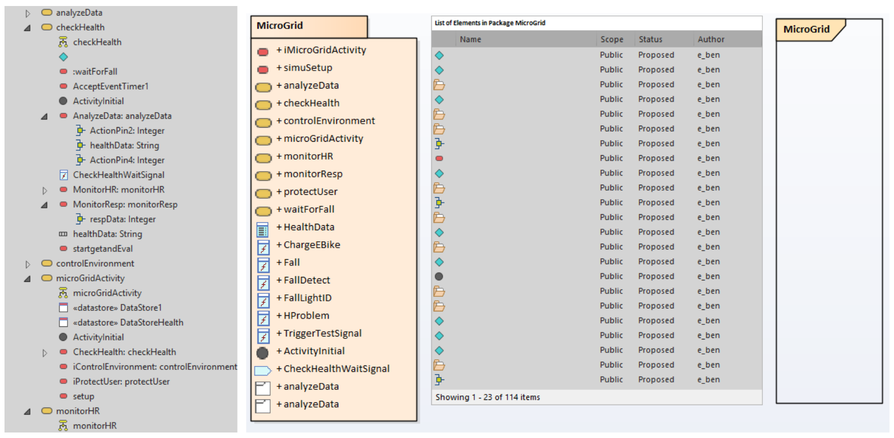
5.2. Scenario

5.2.1. SysML Model

The utilization of SysML for modeling in the context of this use case involves the comprehensive use of Enterprise Architect, a modeling tool renowned for its multifaceted capabilities. While Enterprise Architect offers a wide array of integrated tools, its initial complexity can pose a steep learning curve for users. However, once proficient in its operation, it has proven to be an invaluable asset, facilitating rapid modeling, verification, and validation of complex designs. This use case serves as a platform for designing a SysML model that effectively demonstrates the synergies and interactions between a smart grid and a smart home system. While the use case is deliberately kept straightforward, it readily accommodates expansion, effectively showcasing the advantages of a model-based development approach for such systems. The smart grid within this use case encompasses diverse generator types. These generators exhibit varying characteristics, with some being cost-effective yet environmentally less impactful, while others are costlier but provide higher electricity output. This dichotomy necessitates careful consideration during the modeling process, as it profoundly influences several critical smart grid features. Specifically, this use case defines two generator types: Type A, characterized by higher electricity output at a commensurately higher cost, and Type B, known for its lower electricity capacity but with reduced operational costs compared to Type A generators. Furthermore, the smart grid incorporates renewable energy resources, activating generators only when the available renewable energy sources cannot meet the current electricity demand. In this context, the microgrid represents a smart home environment, equipped with common smart home devices and an Electric Vehicle Charging (EVC) station, adding a layer of complexity and real-world relevance to the model. The SysML model encompasses an array of diagrams, commencing with requirements and package diagrams that serve to organize the model's components. On one facet, the model features behavior and structure diagrams that elucidate the architecture of embedded software, including algorithms, functions, and logic components. On the other facet, the model includes parametric diagrams, which delve into the analytical aspects, representing the physical attributes of the system. Notably, parametric diagrams often require expertise in specific domains, particularly in dealing with physical values, rules, and equations. For instance, when modeling a generator, considerations extend to factors such as the rate at which power is injected into the grid over time, accounting for the startup period during which the generator operates below maximum capacity. This level of detail enables the emulation of the behavior of specific physical components within the system.

Requirements and Package Diagrams

In the realm of IoT and smart grid projects, maintaining a well-structured and comprehensive approach is paramount due to the rapid complexity and scale at which such projects evolve. Iterations in requirements and designs are commonplace as these projects grow. To effectively address these challenges, SysML, specifically package and requirements diagrams, play a crucial role. At the outset of any project, structuring and management are vital. This entails proper allocation of resources, such as time and personnel, and ensuring that both requirements and designs are not only complete but also accurate. In domains like IoT and smart grids, projects can swiftly become intricate and expansive. Consequently, recurrent revisions in requirements and designs are not unusual.



```
▷ analyzeData
◢ checkHealth
    checkHealth
    ◆
    :waitForFall
    AcceptEventTimer1
    ActivityInitial
  ◢ AnalyzeData: analyzeData
      ActionPin2: Integer
      healthData: String
      ActionPin4: Integer
    CheckHealthWaitSignal
  ▷ MonitorHR: monitorHR
  ◢ MonitorResp: monitorResp
      respData: Integer
    healthData: String
    startgetandEval
▷ controlEnvironment
◢ microGridActivity
    microGridActivity
    «datastore» DataStore1
    «datastore» DataStoreHealth
    ActivityInitial
  ▷ CheckHealth: checkHealth
    iControlEnvironment: controlEnvironment
    iProtectUser: protectUser
    setup
◢ monitorHR
    monitorHR
```

MicroGrid

```
+iMicroGridActivity
+simuSetup
+analyzeData
+checkHealth
+controlEnvironment
+microGridActivity
+monitorHR
+monitorResp
+protectUser
+waitForFall
+HealthData
+ChargeEBike
+Fall
+FallDetect
+FallLightID
+HProblem
+TriggerTestSignal
+ActivityInitial
+CheckHealthWaitSignal
+analyzeData
+analyzeData
```

List of Elements in Package MicroGrid

	Name	Scope	Status	Author
◆		Public	Proposed	e_ben
◆		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
◆		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
▣		Public	Proposed	e_ben
⬬		Public	Proposed	e_ben
◆		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
▣		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
◆		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
◆		Public	Proposed	e_ben
●		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
◆		Public	Proposed	e_ben
◆		Public	Proposed	e_ben
◆		Public	Proposed	e_ben
📂		Public	Proposed	e_ben
▣		Public	Proposed	e_ben

Showing 1 - 23 of 114 items

MicroGrid

Figure 5.1.: *package compare*

In this use case, SysML's package and requirements diagrams are harnessed to tackle these challenges effectively. In Figure 5.1, we observe the hierarchical structure of the model in the Project Browser, which can become increasingly unwieldy as the model expands. To mitigate this, various visualizations of a package diagram for the microgrid are presented in the same figure. Enterprise Architect allows the creation of package diagrams for each package, enabling seamless navigation of the entire project. This stands in stark contrast to the Project Browser, which can become unwieldy, especially in large, multi-faceted IoT projects that span numerous domains. In addition to the package diagrams mirroring the project's folder hierarchy (as depicted in Figure 5.2), special package diagrams have been designed explicitly to facilitate navigation in large-scale projects. These diagrams utilize navigation calls independent of the actual folder hierarchy, enabling more natural and straightforward project navigation. Depending on the project's nature and the individuals involved, different package organization structures may be preferred.

In this use case, the primary packages include "Structures," which contains links to all the structure diagrams categorized based on diagram type, "UseCases," and "Requirements," all organized similarly to the "Structures" package. Additionally, the "Activity Simulation Scenarios" package diagram links to significant activities used for demonstration with SysMLSim (as seen in Figure 5.4). Lastly, the "Dimension and Units" package links to the folder named "Dimensions and Units." Enterprise Architect suggests various libraries

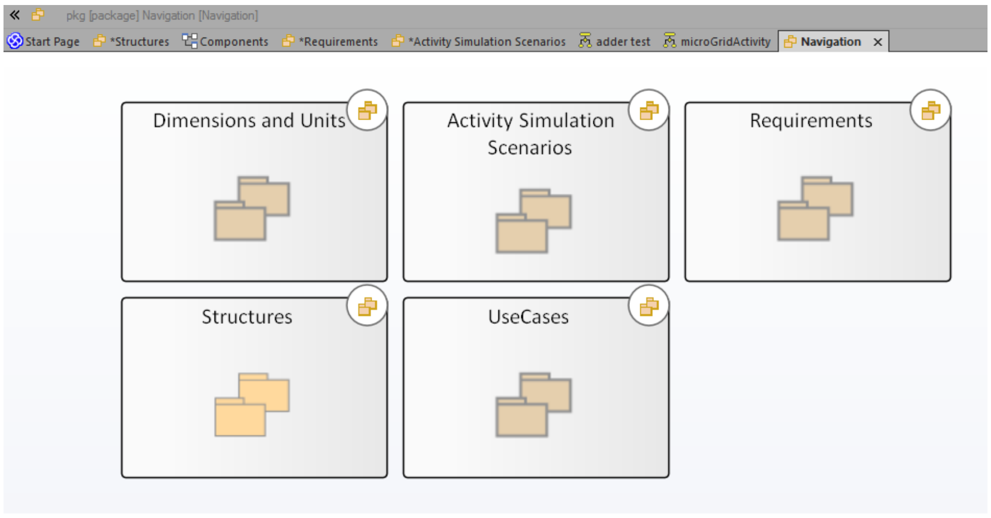


Figure 5.2.: *package project navigation*

containing commonly used elements in software engineering when creating a project. One such library is "Dimension and Units," defining widely used units across various domains, including physics units (e.g. kilograms and seconds) and electrical engineering units (e.g. Amperes for electric current), as illustrated in Figure 5.3.

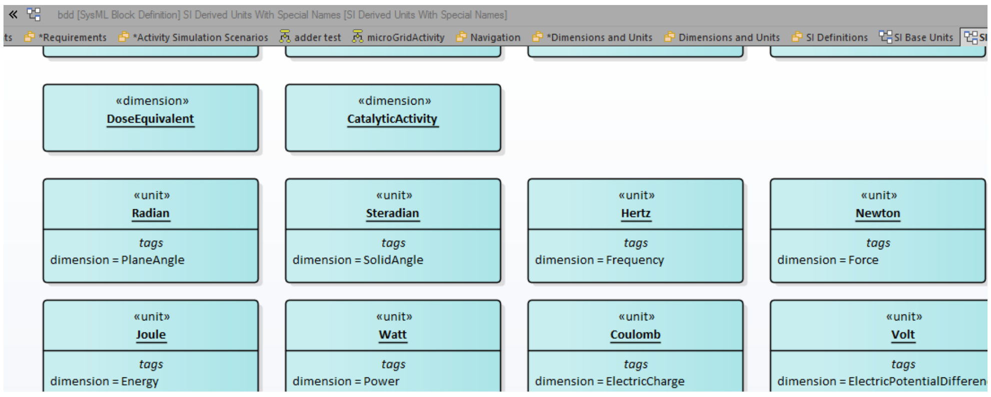


Figure 5.3.: *package units*

Subsequently, a crucial step involves establishing requirements. For the smart home, fundamental functional requirements have been defined, such as the activation of lights upon motion detection (as presented in Figure 5.5). In contrast, requirements for the smart grid are framed in a more general manner, emphasizing self-healing properties and dynamic pricing. This generality is advantageous, as SysML accommodates the creation of high-level requirements, (refer to Figure 5.5).

SysML's capability to allocate requirements to functions, functions to structures, and activities to operations is instrumental in maintaining traceability. When a requirement undergoes modification, it becomes straightforward to

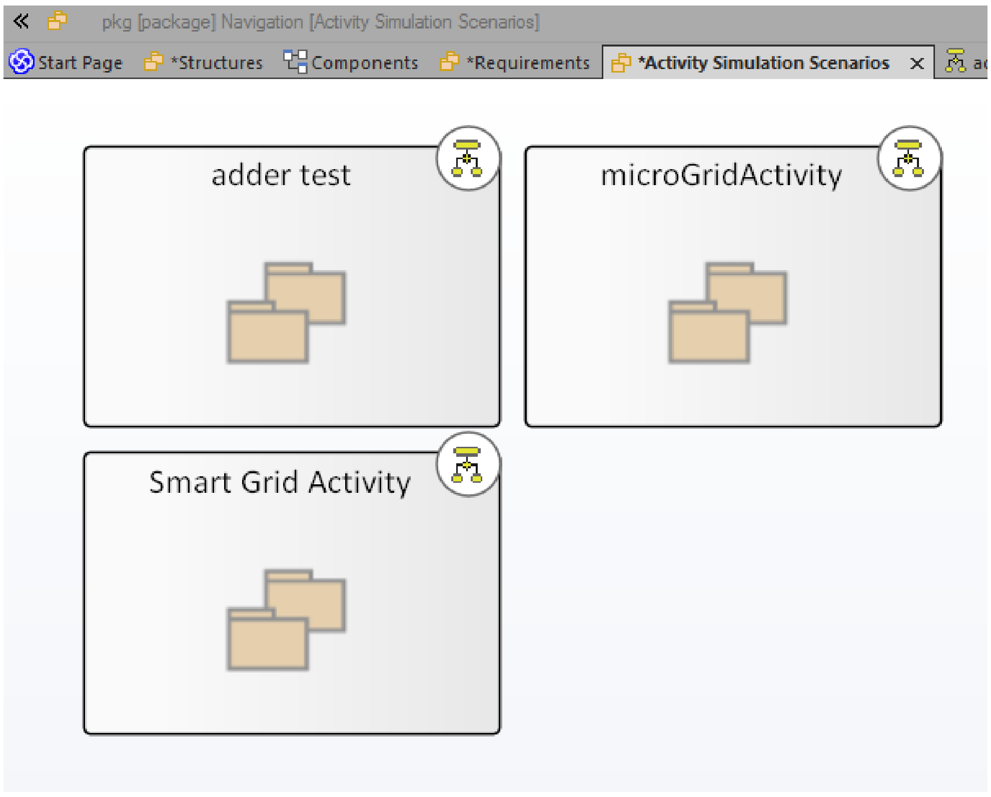


Figure 5.4.: *package sim scenarios*

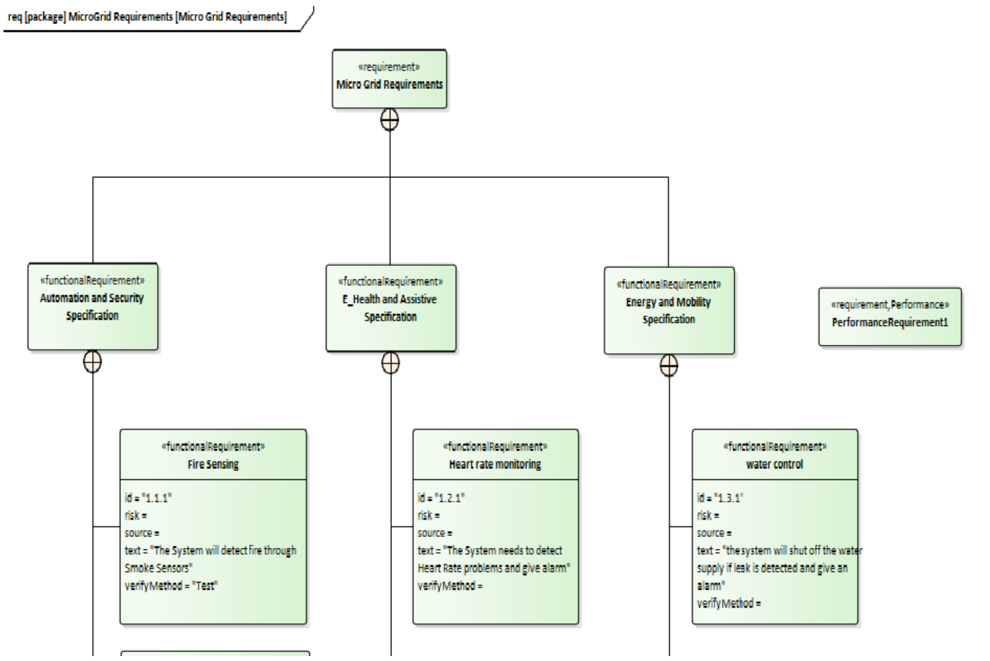

req [package] MicroGrid Requirements [Micro Grid Requirements]
«requirement» Micro Grid Requirements
«functionalRequirement» Automation and Security Specification
«functionalRequirement» E_Health and Assistive Specification
«functionalRequirement» Energy and Mobility Specification
«requirement,Performance» PerformanceRequirement1
«functionalRequirement» Fire Sensing
id = "1.1.1"
risk =
source =
text = "The System will detect fire through Smoke Sensors"
verifyMethod = "Test"
«functionalRequirement» Heart rate monitoring
id = "1.2.1"
risk =
source =
text = "The System needs to detect Heart Rate problems and give alarm"
verifyMethod =
«functionalRequirement» water control
id = "1.3.1"
risk =
source =
text = "the system will shut off the water supply if leak is detected and give an alarm"
verifyMethod =

Figure 5.5.: *Requirements Diagram for the micro grid*

track all associated models and revise them accordingly. Likewise, when a model is altered, it is easy to trace all the affected requirements. The package diagrams and requirement diagrams provide robust traceability throughout the development process.

Block Definition Diagram and Internal Block Diagram

The Block Definition Diagram (BDD) and Internal Block Diagram (IBD) play a crucial role in defining the structure and composition of complex systems, enabling engineers to visualize and specify the relationships between different components and subsystems. In this use case, these diagrams are employed to provide a clear and detailed representation of both the smart grid and the micro grid, which is essentially a smart home with various subsystems and devices.

Block Definition Diagram (BDD) The Block Definition Diagram, as shown in Figure 5.6, serves as a high-level overview of the micro grid within the smart home. It defines the key subsystems, their associated devices, and their interconnections. The subsystems are based on the VICINITY smart home test lab, comprising Home Automation and Security, Energy Consumption and E-Mobility, and E-Health and Assisting Technologies. These subsystems are encapsulated blocks that contain attributes, operations, and relationships.

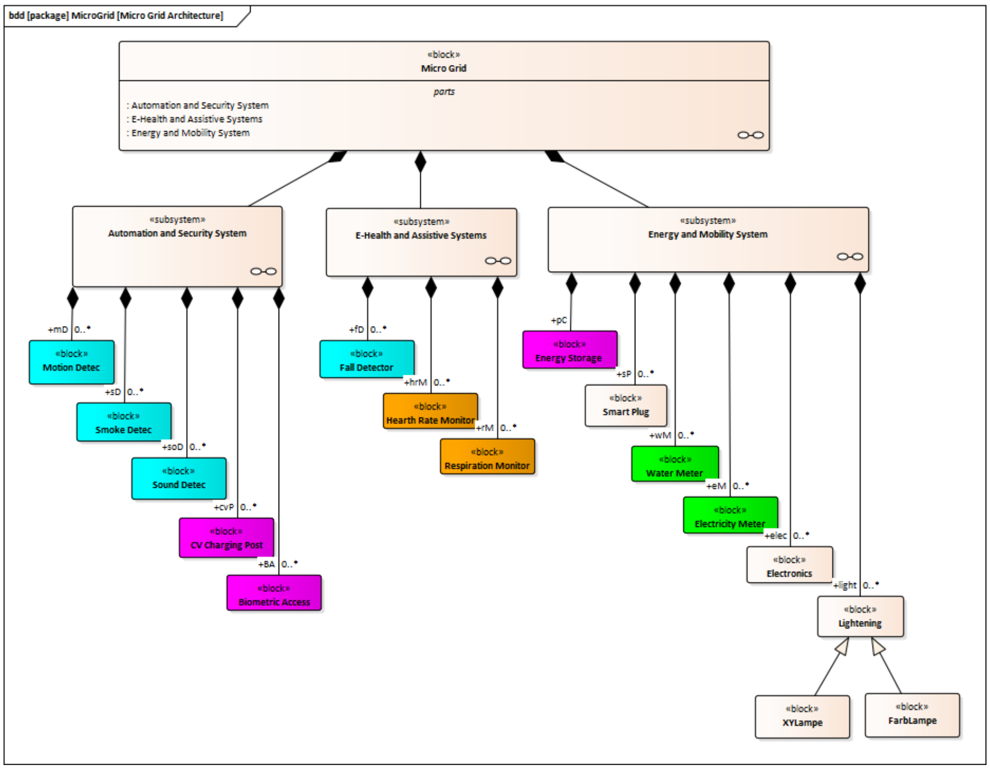


Figure 5.6.: *Micro Grid Block Diagram*

Among the most important devices in this use case are the fall sensor, motion sensor, E-Bike charging station, heart rate, and respiration monitors. These devices are part of the Home Automation and Security and E-Health and Assisting Technologies subsystems. Additionally, the energy storage and electricity meter are pivotal components for the smart grid, providing essential information about energy consumption. The smart grid, represented in a simplified manner, is composed of micro grids and various generators. Future iterations of this model could introduce more complexity by including different types of micro grids or additional generators. For this use case, a single micro grid, which represents a smart home, is considered.

Internal Block Diagram (IBD) The Internal Block Diagram, exemplified in Figure 5.7, delves into the intricate interactions between objects within each component. It elucidates how objects flow from one element to another, offering a comprehensive view of data exchanges and connections.

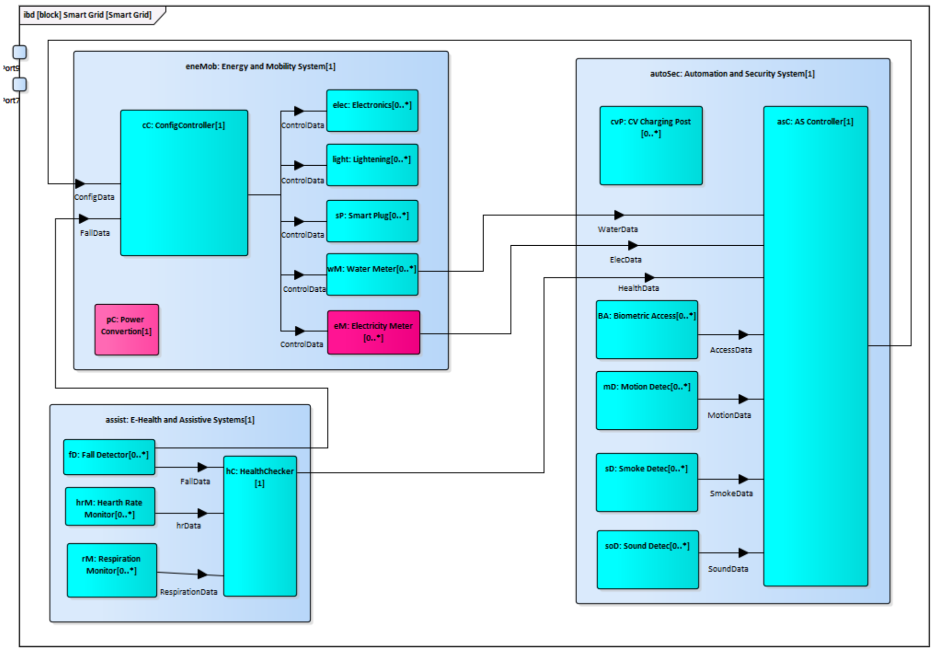


Figure 5.7.: *Micro Grid Internal Block Diagram*

In Figure 5.7, the flow of objects within the micro grid is illustrated. It showcases the paths through which objects pass between various components, providing insights into how electricity, water, and other resources are transferred between systems. In the appendix, each component's IBD is detailed, enumerating the ports through which objects are transmitted. Together, the BDD and IBD are instrumental in modeling the structure and functionality of the micro grid and smart grid in this use case. They help in understanding how subsystems are organized, how devices interact, and how data and resources are shared within the system. This level of detail is crucial for accurate system design, analysis, and validation, especially in the context of complex IoT and smart grid projects where precision and clarity are paramount.

State Machines and Activity Diagrams

The State Machine and Activity Diagrams are fundamental tools in SysML for modeling the dynamic behavior and activities of a system. In this use case, they are utilized to depict various operational states and processes within both the smart home (micro grid) and the smart grid.

State Machine Diagrams State Machine Diagrams, exemplified by Figure 5.8 and Figure 5.9, are employed to represent the states and transitions of devices and systems. In these diagrams, simplicity is maintained by primarily focusing on the on/off states of devices. However, for more intricate models, transitions between states can be triggered by specific events, such as signals or external inputs.

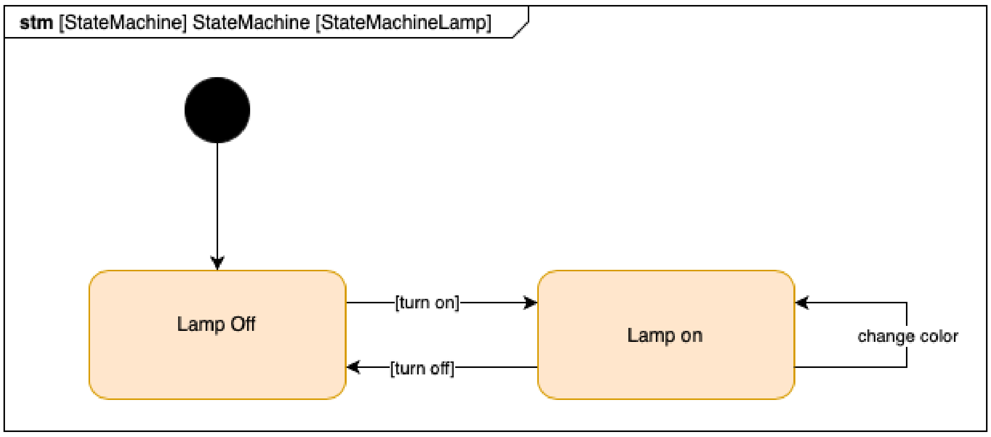


Figure 5.8.: *State Machine of the lighting behaviour*

As an example, consider the generator in Figure 5.9. It transitions from the ”on” state to the ”powering down” state upon receiving a ”generator off” signal. To account for the time required for generators to power up or down, additional states are introduced, each of which triggers corresponding behaviors or activities defined in the Activity Diagrams (e.g. ”generator off” and ”generator on”).

Activity Diagrams Activity Diagrams, demonstrated in Figure 5.10, offer a more detailed view of the micro grid (smart home) and its three interacting subsystems: Home Automation and Security, Energy Consumption and E-Mobility, and E-Health and Assisting Technologies. These diagrams illustrate the flow of activities and interactions between subsystems.

Moreover, the Activity Diagrams reveal how the micro grid interacts with the smart grid, providing insights into their combined workflow and demonstrating the potential benefits of integrating and modeling these systems together.

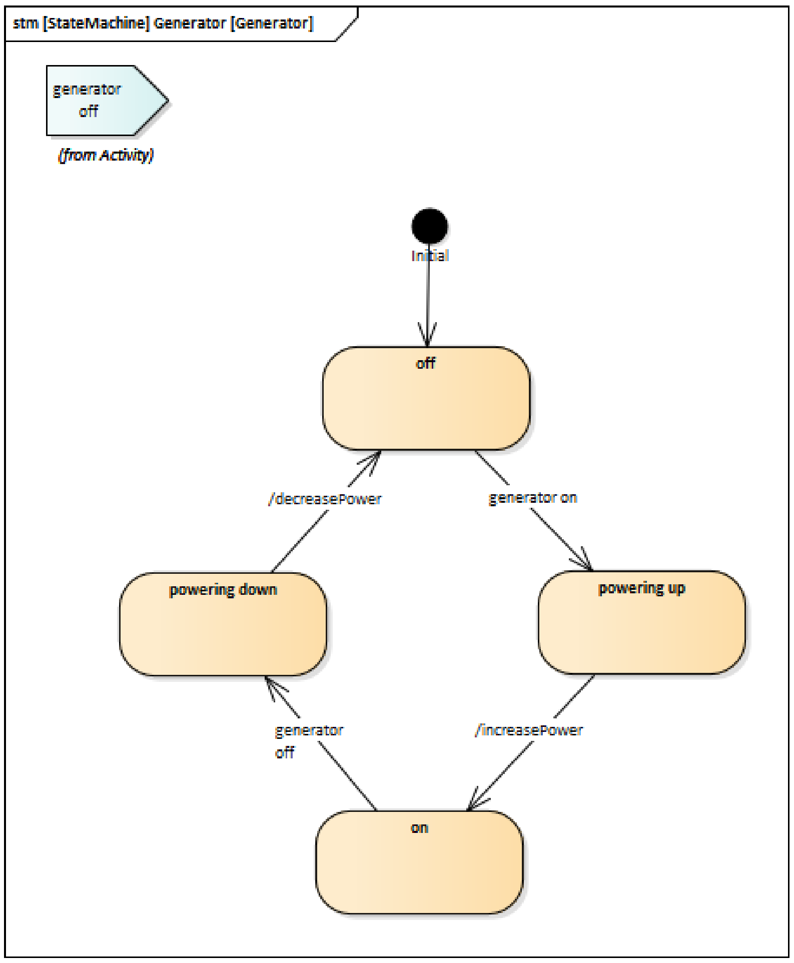


Figure 5.9.: *State Machine of a generator*

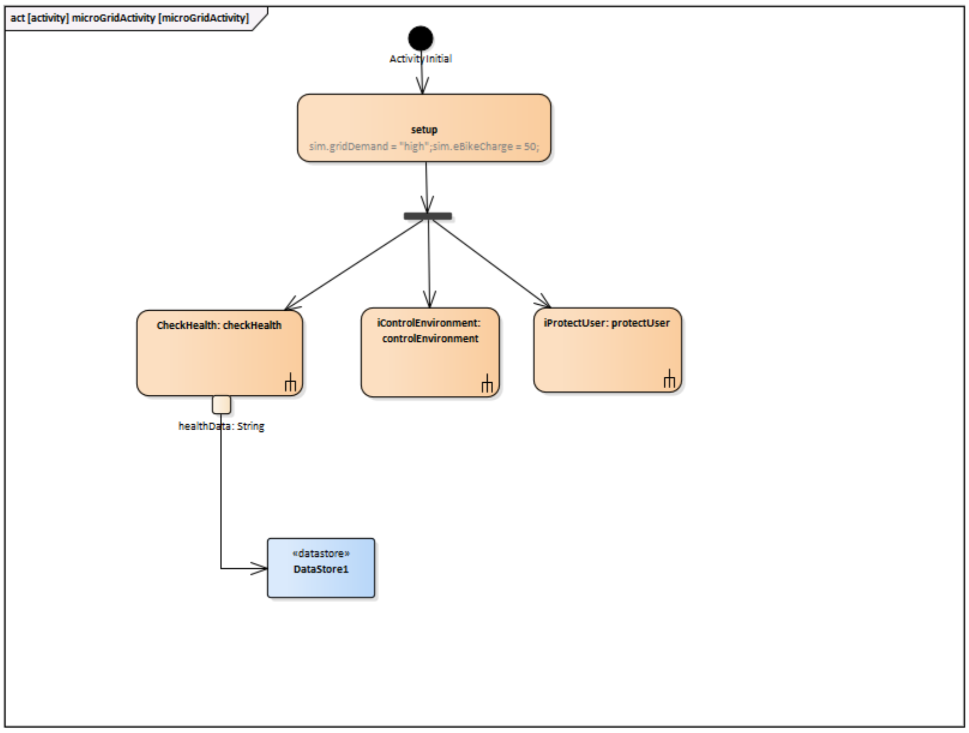

act [activity] microGridActivity [microGridActivity]
ActivityInitial
setup
sim.gridDemand = "high";sim.eBikeCharge = 50;
CheckHealth: checkHealth
iControlEnvironment: controlEnvironment
iProtectUser: protectUser
healthData: String
«datastore»
DataStore1

Figure 5.10.: *Activity Diagram of the Micro Grid*

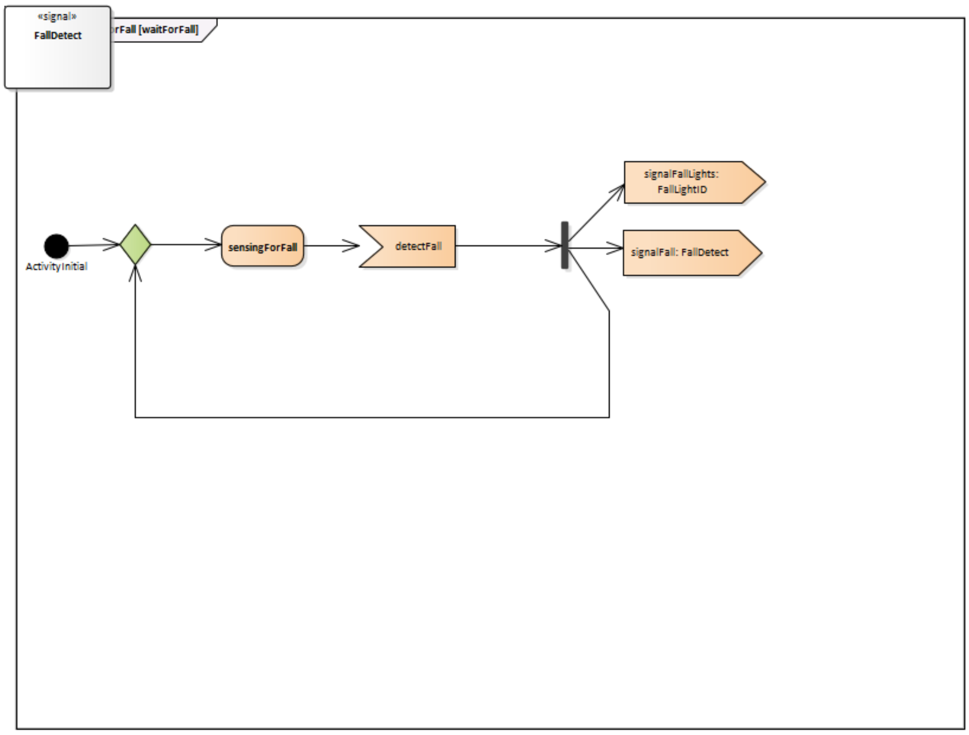


Figure 5.11.: *Fall sensor sensing Activity Diagram*

In these diagrams, various behaviors and trigger events common in a smart home environment are modeled. These triggers can be simple signals originating from devices or external sources (as seen in Figure 5.11). For example, when charging an e-bike, it can increase the electricity consumption in the micro grid, which triggers a generator in the smart grid to power up when electricity prices are low.

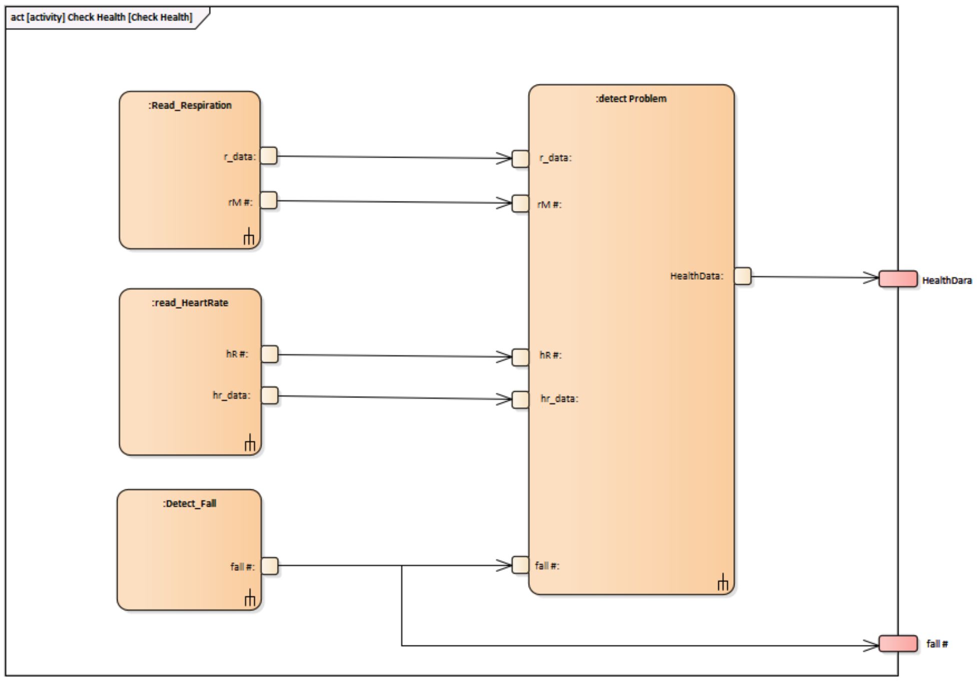


Figure 5.12.: *Check Health Activity Diagram*

The flexibility of SysML's trigger events allows for the modeling of various scenarios. For instance, in the "CheckHealth" diagram (see Figure 5.12), a fall and motion detector are modeled to respond to specific trigger events. When a fall is detected, signals are sent to activate the lighting and initiate a health evaluation. These diagrams depict how devices react to real-world situations, making it possible to simulate complex interactions within the system. These diagrams provide a simplified view of complex systems. In more advanced simulations, real-world data and sophisticated evaluation algorithms could be incorporated. For instance, health monitoring systems could analyze data trends over time and issue alerts based on more advanced criteria. Similarly, the smart grid model could consider various factors, such as equipment failures or sudden spikes in demand, in a more comprehensive simulation environment. Specialized simulation tools would be better suited for such high-fidelity simulations. In summary, the State Machine and Activity Diagrams in SysML are invaluable for modeling dynamic behaviors and processes within complex systems, making them an essential tool for IoT and smart grid projects. They allow for the visualization of different scenarios, event triggers, and complex interactions, contributing to a better understanding of system dynamics and behavior.

Preliminary Conclusions

In this subchapter, we have explored the application of SysML (Systems Modeling Language) to model and analyze a complex use case involving IoT, smart grids, and smart homes. SysML, as a powerful modeling language, offers a range of diagrams and tools that facilitate the structured representation of system architecture, behavior, requirements, and more. We began by emphasizing the importance of proper project management, resource allocation, and the need for complete and accurate requirements in the context of IoT and smart grid projects. SysML's package diagrams and requirements diagrams provided an efficient means to organize and manage these requirements. We demonstrated how SysML aids in handling complex projects by creating package diagrams that help navigate extensive project structures effortlessly. Moving forward, we delved into the Block Definition Diagram (BDD) and Internal Block Diagram (IBD). BDDs enabled the definition of system composition and parts, while IBDs showcased the flow of objects among different system components. In our use case, we examined a simplified smart grid and a detailed smart home micro grid. The IBDs illustrated how objects, representing various forms of resources such as electricity and water, moved within and between these components. State Machine Diagrams and Activity Diagrams provided insights into system behavior. State Machine Diagrams primarily focused on simple device states and transitions, demonstrating how devices could be turned on and off or undergo other state changes. Activity Diagrams, on the other hand, depicted complex interactions and workflows within the smart home and between the micro grid and smart grid. Various trigger events were modeled to simulate real-world scenarios, showcasing the flexibility of SysML in capturing dynamic behavior. This subchapter and the SysML models generated generously answers **question 1** and **question 2** from the target setting of this dissertation (refer to chapter 1.4.1). In conclusion, SysML serves as a valuable tool for modeling and simulating intricate systems like IoT-enabled smart grids and smart homes. It enables efficient project management, requirement organization, system composition definition, and behavior modeling. While our use case provided simplified models, SysML can accommodate more extensive and detailed representations for higher-fidelity simulations and analysis. In the realm of IoT and smart grids, where complexity is inherent, SysML empowers engineers and designers to create structured, comprehensive, and adaptable models, facilitating better system understanding and decision-making throughout the project lifecycle.

5.2.2. Smart City

In order to exemplify the practical application and the efficacy of the multifaceted simulator developed within this study, we have meticulously modeled and simulated a smart energy use case that incorporates homomorphic encryption. This use case amalgamates elements derived from prior works, specifically, [87] and [88]. The focal point of this particular use case resides in the realm of smart energy management within an urban setting. Within the simulated cityscape, several integral components interplay to create a dynamic smart energy ecosystem. The city itself serves as the central stage, boasting a renewable energy infrastructure comprised of photovoltaic arrays and wind turbines that act as power supply sources. Concomitantly, the urban landscape features a parking facility and a collection of residential houses, serving as energy consumers. This dynamic energy landscape is further enriched by the presence of electric vehicles (EVs) that navigate the city's thoroughfares and frequent the parking facility. A pivotal element of this smart energy ecosystem is the implementation of a sophisticated smart parking service, operable through a dedicated mobile application. This service empowers system users to solicit the reservation of specific parking slots within the participating parking facilities, enhancing convenience and efficiency in urban commuting. The real-time availability status of these parking slots is diligently relayed to users via the mobile application interface. Additionally, optical indicators stationed at the respective parking slots provide visibility to individuals who do not engage with the smart parking service. To comprehensively evaluate the efficacy of our developed simulation framework, we have instantiated and scrutinized this complex smart city scenario at three distinct levels of abstraction. The initial two higher levels of abstraction are exclusively implemented using classes inherent to the core simulator developed within this dissertation. In contrast, the third and most granular level is realized by leveraging OMNeT++ 5.4.1 in conjunction with its INET extension 4.0. This multi-tiered approach facilitates a comprehensive evaluation of the smart parking scenario's performance, offering insights into the system dynamics at varying levels of detail.

The highest abstraction level - Level 0

At the pinnacle of the abstraction hierarchy, the highest level, denoted as Level 0, encapsulates the modeling of fundamental and abstract processes. These processes are instrumental in furnishing essential information that cascades down to the ensuing lower levels, thus priming the simulation scenario with the requisite foundational data. Notably, at this lofty level, the power-generating entities within the smart city scenario find their representation. This overarching abstraction is elegantly illustrated in Figure 5.13.

The focal entity at this zenith of abstraction is the CarGenerator atomic model. Functioning as a fount of data and events, the CarGenerator is tasked with delivering vital information that underpins the lower-tier simulations. This information serves the dual purpose of simulating both users and random vehicular arrivals at the forthcoming levels of abstraction. The conduit for this data transfer is the CarProcessor. The CarProcessor, residing at the apex of this abstraction hierarchy, plays a pivotal role in categorizing and channeling the incoming data streams. It segregates the information into distinct categories, crucially distinguishing between scenarios involving random visitors to the parking facility and users of the smart parking mobile application.

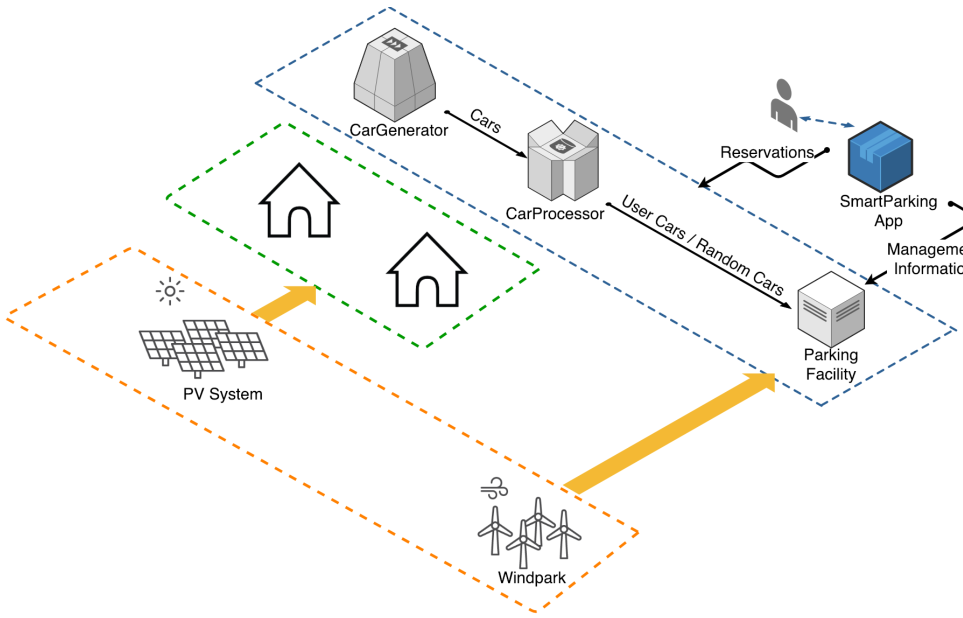


Figure 5.13.: *Smart Energy use case: Level 0*

In the latter instance, the CarProcessor orchestrates the generation of parking slot preferences, mimicking the user's intent to reserve a specific parking space through the model of the smart parking application. If this reservation endeavor proves unsuccessful, the data characterizing the app user's preferences is transmuted into the semblance of information about a random visitor to the facility, which is then subsequently conveyed to the ParkingFacility model. Upon traversing into the precincts of the ParkingFacility model, the received data takes on a pivotal role in crafting an abstract representation of the dynamic interplay between random visitors and users of the smart parking service. These entities engage in a virtual contest for the limited parking slots, engaging in activities such as parking and exiting the facility. For users of the smart parking application who have successfully secured a reservation, their path leads directly to their designated parking slots. Conversely, random arrivals and users who encountered reservation failures embark on a quest for the first available parking space. Upon reaching a parking slot, a rigorous assessment is undertaken to determine its current status - whether it remains vacant or has been reserved by another user or claimed by an earlier-arriving random vehicle. In the event that the slot has already been occupied, the searcher will navigate back to the pursuit of an unoccupied parking space. Failure to secure a parking space within a reasonable time threshold will result in the vehicle exiting the parking facility. Conversely, a triumphant parking endeavor culminates in the vehicle temporarily occupying the chosen parking slot, followed by a subsequent exit from the facility. This intricate dance of arrivals, reservations, and parking events unfolds within the richly abstracted landscape of Level 0.

The middle abstraction level - Level 1

Advancing to the intermediary stratum of abstraction, denoted as Level 1, our focus shifts to a more granular representation of the internal processes operating within the ParkingFacility. This tier, as elucidated in Figure 5.14, introduces an intricate division of the ParkingFacility into three discrete parking decks, each of which internally mirrors the operational dynamics of the parking facility situated in Tromsø. The essential framework at Level 1 revolves around the meticulous orchestration of car arrivals, their traversal within the facility, and eventual departures. Incoming data pertaining to vehicle arrivals is systematically relayed to the individual parking decks in a sequential manner. This orchestration becomes particularly significant when vehicles enter the facility, for they must navigate the multi-tiered parking decks until they locate their desired parking slot. Conversely, during a vehicle's egress from the ParkingFacility, it must again traverse these decks to reach the exit point. The linchpin of this operational paradigm is the ParkingDeckControl, a critical entity tasked with the bi-directional flow of information. It acts as the lynchpin for the exchange of data between the smart parking application's model and the intricate layers of the parking facility simulation. This two-way conduit ensures that the information is effectively transmitted to the relevant parking deck for processing, orchestrating the allocation and utilization of parking slots in accordance with user preferences and facility conditions.

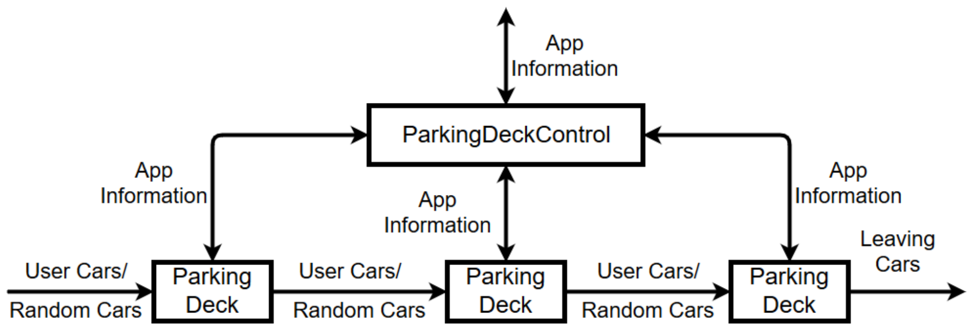


Figure 5.14.: *Smart Energy use case: Level 1 - the Parking Facility*

The lowest abstraction level - Level 2

The culmination of our simulation scenario resides at the lowest tier, Level 2, where meticulous attention to detail is paramount. This stratum has been meticulously crafted utilizing OMNeT++ 5.4.1 and INET 4.0, enabling a dynamic instantiation of simulated entities and the facilitation of communication channels between sensors, actuators, and the mobile application, harnessing the advanced capabilities endowed by INET. At this level, the parking facility manifests as an OMNeT++ ”compound module,” with each individual parking deck represented as submodules thereof. One such parking deck modeled in OMNeT++ can be found in Figure 5.15. Notably, while interactions can transpire among these submodules across their boundaries, only those components linked to the active segments of the higher-level simulation are functionally operational. When the upper-level parking deck model transitions to the pertinent segment of the OMNeT++ module in Level 2, a symphony of mobile nodes, the cars, materializes. These vehicles assume distinctive characteristics dictated by the information furnished from the higher echelons. The behavioral dynamics of these cars are meticulously orchestrated by the states and conditions defined at Level 1. Depending on their specific phase – whether they are actively seeking a vacant parking slot, are parked, or are poised to exit the parking deck – the corresponding wireless nodes are instantiated, and their objectives are aligned accordingly. Each car is endowed with a battery, which depletes while the vehicle is in motion within the simulated environment. Upon parking within the facility, the car's battery enters a recharging phase, steadily refilling its accumulator. When the battery reaches full capacity, the charging ceases, signifying that the vehicle is primed for departure, duly equipped to navigate the digital urban terrain. This intricate interplay of mobile entities and their energy dynamics is emblematic of the attention to detail manifest at this lowest level of abstraction, demonstrating the sophistication of our simulation framework.

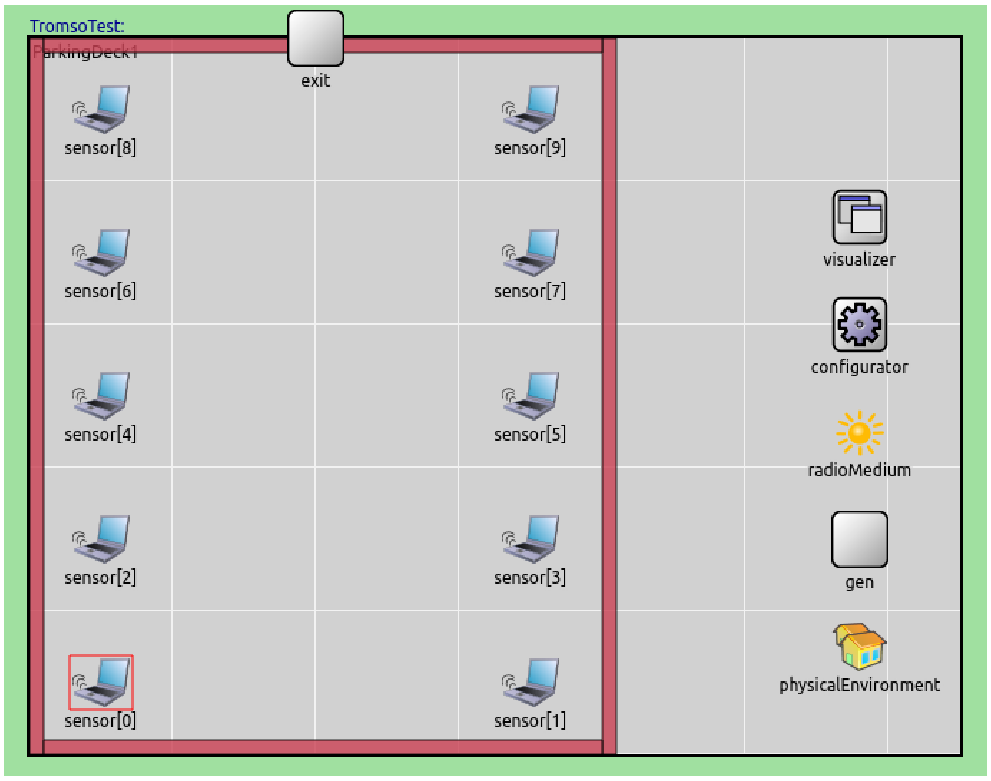


Figure 5.15.: *Smart energy use case: Omnet++ Model of Parking Deck at Level 2*

VICINITY Bridge

Intricately intertwined with our simulation framework's fabric is the VICINITY Bridge, a pivotal element that showcases the profound integration of our hardware-in-the-loop interface with the actual VICINITY network. As discussed comprehensively in Section 2.7, VICINITY stands as a robust infrastructure, orchestrating the connection of disparate IoT ecosystems that would otherwise exist as isolated islands. It is worth noting that our meticulously crafted simulated network can indeed be categorized as one such island, with its digital devices existing solely in the realm of simulation. The introduction of a VICINITY adapter into our Omnet++ simulation, as delineated in Section 4.7.2, facilitates this seamless integration with VICINITY, thereby eliminating any perceptual boundaries between our simulated network's virtual devices and VICINITY's real-world devices and value-added services. To VICINITY, the devices in our simulation, existing purely in the digital realm, appear as ordinary entities, indistinguishable from their physical counterparts. Simultaneously, to our value-added services, these virtual entities exhibit analogous characteristics to genuine physical devices. This inherent fusion of virtual and physical realms endows us with a multitude of advantages:

- **Rapid Prototyping:** We are empowered to swiftly prototype and scrutinize diverse scenarios or use cases without the necessity of an initial physical deployment, accelerating the development cycle significantly.
- **Scalability Testing:** The ability to effortlessly scale up existing use cases and assess their behavior with an expanded fleet of attached devices equips us with invaluable insights into scalability aspects.

Given the considerable expense and time investments required for the installation and deployment of new sensors on our parking deck, as elaborated in Figure 5.14, this virtual setup serves as an agile and cost-effective means to commence and continue the development of our front-end application. It operates in parallel with the real-world deployment, expediting progress. Furthermore, this virtual environment affords us a unique vantage point for evaluating diverse approaches concerning our enhanced privacy modules, which are underpinned by homomorphic encryption. In the ensuing sections, we harness this virtual framework to conduct empirical measurements and assess the real-time impact of various homomorphic encryption schemes on the overall system runtime in a live test scenario, shedding light on their practical implications.

Integration of the homomorphic encryption micro-service

As expounded upon in Section 2.8, homomorphic encryption emerges as a potent tool when sensitive data must undergo processing by an entity whose trustworthiness is not unequivocal. This scenario often arises when users expect specific benefits from a third-party value-added service or when compelled to employ such a service, prompting concerns regarding the confidentiality of their data. The absence of trust can manifest as a formidable impediment in the domain of the Internet of Things (IoT). Homomorphic encryption proffers an elegant solution to these privacy quandaries. However, the sanctuary of privacy it bestows carries the associated cost of heightened computational demands, encompassing encryption, decryption, and the execution of functions on ciphered data, all of which exact an augmented computational toll. Within the VICINITY project (refer to Section 2.7), homomorphic encryption promised enhanced data security and privacy assurance. In the pursuit of pragmatic viability, it becomes imperative to scrutinize the computational overhead incurred by the integration of homomorphic encryption. Furthermore, the profusion of potential encryption schemes, each offering unique homomorphic properties, necessitates comparative evaluation. This evaluative process is instrumental in distinguishing between partially and fully homomorphic encryption schemes, elucidating the computational overhead introduced by the latter, and benchmarking both against a scenario bereft of encryption measures, thus void of privacy safeguards at the value-added service (VAS) level. To this end, we embarked on a comprehensive simulation effort, mirroring a representative use case within VICINITY, facilitated by partially homomorphic encryption, fully homomorphic encryption, and a non-encrypted scenario. This rigorous analysis of computational costs was performed under controlled laboratory conditions, harnessing Hardware-in-the-Loop simulations of the VICINITY infrastructure, affording rapid and cost-effective adjustments compared to physical re-deployment on site. Visual representations of the two simulated use cases can be gleaned from Figures 5.16 and 5.17, each elucidating distinctive facets of the privacy-preserving mechanisms at play.

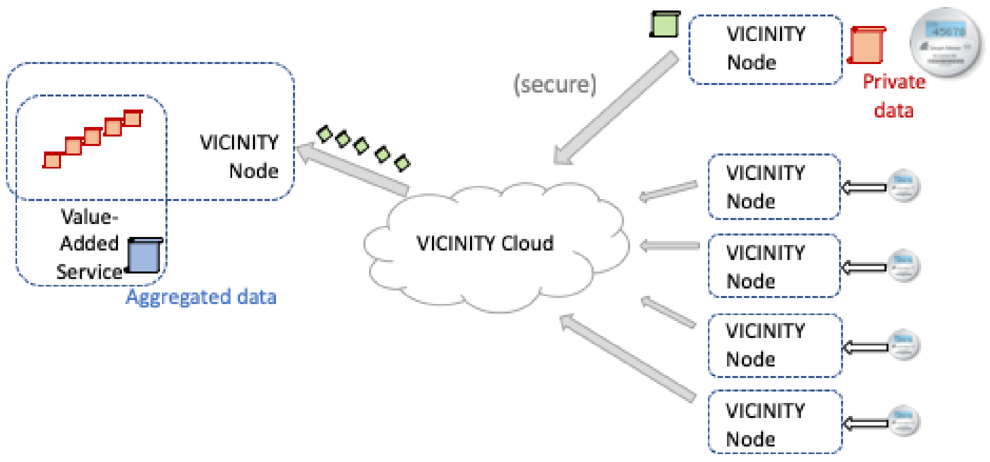


Figure 5.16.: *Use case integration into the VICINITY network. Value-added services have clear text access to private information.*

As depicted in Figure 5.16, data pertaining to energy consumption from the simulated vehicles is amassed and transmitted to the operator's VAS. The operator's sole interest lies in the collective energy consumption of the entire vehicle fleet, thereby initially computing the sum of all inputs. Owing to the potential privacy implications associated with deducing behavioral patterns from individual energy consumption data, this data category is designated as

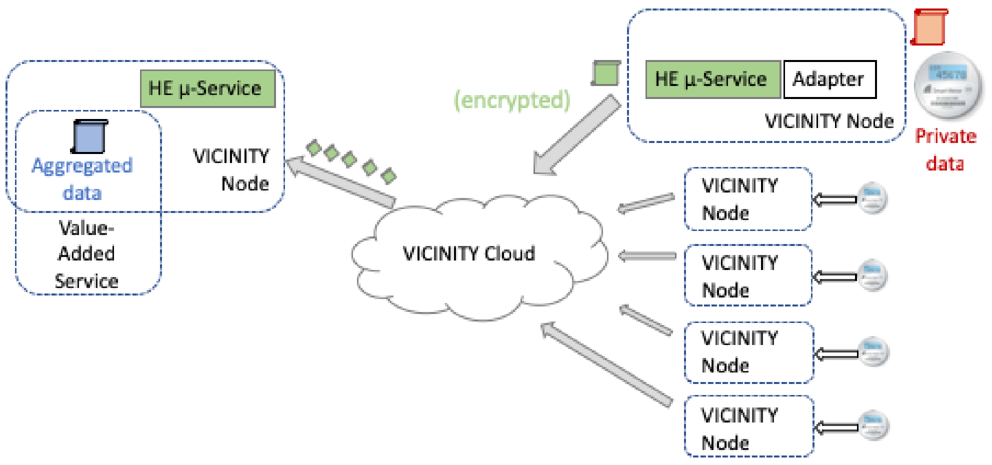


Figure 5.17.: *Homomorphic encryption micro-service applied to use case*

confidential and its non-disclosure is paramount. Given the lack of incentive for the operator or the users to divulge their individual private data, the homomorphic encryption micro-service is introduced, seamlessly integrating into the VICINITY dataflow, as showcased in Figure 5.17. This configuration is juxtaposed with the setup in Figure 5.16. In the new arrangement, input data (e.g. the energy consumption of each vehicle) undergoes encryption using a homomorphic encryption scheme. In the context of partially homomorphic encryption, a simple Paillier encryption scheme[96], capable of cipher text addition, suffices to calculate the aggregate energy consumption of the entire fleet. Considering the potential necessity of more complex operations in future use cases, we also simulated this scenario using the Brakerski-Gentry-Vaikuntanathan (BGV) scheme[97], exemplifying a fully homomorphic encryption scheme. In both instances, the encrypted payload is subsequently transmitted through the VICINITY peer-to-peer network and delivered to the operator's value-added service. The homomorphic encryption micro-service engages in cipher text addition on the encrypted inputs, followed by decentralized decryption of the aggregated data. In this manner, only anonymized, consolidated data is presented and furnished to the value-added service, ensuring the sanctity of user privacy. All measurements and evaluations are juxtaposed with the baseline use case represented in Figure 5.16, offering a comprehensive assessment of the computational overhead incurred in the quest to fortify data privacy, and its ramifications in a real-world context.

5.2.3. Smart Home

The smart home scenario we present here builds upon the foundation established in the preceding smart city scenario detailed in Section 5.2.2. This scenario, initially designed for modeling a parking facility comprised of three parking decks with capabilities such as car entry, parking, and charging, is augmented to encompass the intricacies of a smart home environment. The smart home scenario involves the integration of a simulated house environment, derived and customized from the presented SysML model. These simulated houses comprise a multitude of components, including:

- **Electronic Devices:** This category encompasses a diverse range of devices, such as washing machines, computers, and others, each contributing to the overall energy dynamics of the household.
- **Light Control:** This facet enables the management and control of lighting within the smart home, introducing an additional layer of energy management and user convenience.
- **Security Control:** The security control component encompasses multiple aspects, including intrusion detection against burglars and fire detection, fortifying the safety of the occupants.
- **Health Control:** The health control system is a multifaceted element within the smart home scenario. It encompasses the ability to detect falls, monitor heart rate, and track respiration, all of which are crucial for ensuring the well-being of the inhabitants.

In the context of this simulation, each house is occupied by a single individual who follows a daily routine. On working days, this individual departs for work, with the precise timing governed by randomization. During the individual's absence, electronic devices within the house remain dormant, reflecting real-world patterns of usage. Crucially, the simulation imposes constraints such that electronic devices only operate when the house's occupant is present. Randomly selected electronic devices activate and begin functioning when the individual is at home, thereby contributing to the energy dynamics of the household. Moreover, throughout the simulation, health and security control systems continuously gather data and respond to any deviations from expected norms. These systems are instrumental in alerting the house's occupant to any anomalies or potential issues related to health, safety, or security. The energy consumed by electronic devices, lighting, and the various control systems collectively constitutes the overall energy consumption of the smart home. This complex, multi-faceted simulation is designed to replicate real-world dynamics and interactions within a modern smart home environment.

High-Level Model Overview

As elucidated in the preceding section, the smart home scenario is meticulously implemented through a multi-level simulation approach within the OMNeT++ framework. A comprehensive class diagram encapsulating the entire scenario provides a high-level overview, as depicted in Figure 5.18.

In this multi-tiered simulation framework, each layer plays a distinct and critical role in replicating the intricate dynamics of a modern smart home environment. Below, we provide an overview of each of these layers and their constituent components:

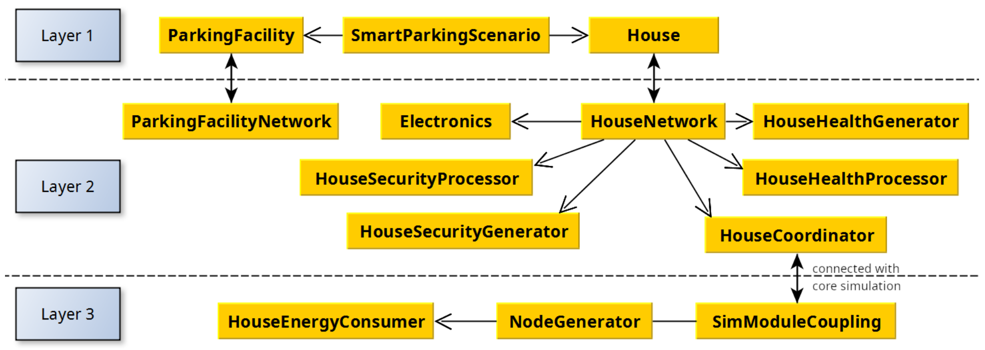


Figure 5.18.: *Class diagram as an overview of the scenario*

Layer 1 - The Foundation At the lowest abstraction level, denoted as Layer 1, resides the fundamental building block - the House. This House forms an integral part of the broader SmartParkingScenario, serving as the bedrock upon which the entire simulation is constructed. Within this layer, the House coexists with the ParkingFacility and the Windpark, each playing a role in shaping the energy dynamics and the environment of the simulation.

Layer 2 - Augmentation and Extension Layer 2 introduces a layer of complexity and refinement to the House by extending it with the HouseNetwork. This layer follows a hierarchical approach, with the House serving as the atomic model, while the HouseNetwork takes on the role of a network model positioned one layer above the atomic model. Consequently, the HouseNetwork and its associated atomic models effectively augment the capabilities of the House as envisioned in Layer 1.

The pivotal atomic models at this layer include:

- **HouseSecurityGenerator:** Responsible for generating security events, such as fire or intrusions, which are subsequently communicated to the *HouseSecurityProcessor* for further processing.
- **HouseSecurityProcessor:** Receives security events generated by the security generator and responds by triggering alarms within the *House*. These alarms, representing potential fire or intrusion incidents, remain active for a specified duration before being automatically deactivated.
- **Electronics:** This module serves as the manager of electronic devices within the simulated household. It makes decisions regarding when to activate or deactivate electronic devices, providing this information to the energy management system for further analysis and control.
- **HouseHealthGenerator:** Responsible for generating health-related events, specifically coronary events, which are then transmitted to the *HouseHealthProcessor*.

- **HouseHealthProcessor:** Tasked with analyzing health events for abnormalities, the health processor dispatches health alarms to the *House* whenever deviations from normal health parameters are detected.
- **HouseCoordinator:** Acting as a central hub, the house coordinator aggregates and disseminates events from various modules within this layer. It serves as the linchpin for communication between Layer 2 and Layer 3. Events received from Layer 3 are routed to the appropriate module within Layer 2, facilitating seamless information flow.

Layer 3 - Orchestrating the Simulation The third layer, represented by the module SimModuleCoupling, plays a pivotal role in orchestrating the simulation. It acts as an interface between the layers, converting outputs from the modules in Layer 2 into OMNeT++ messages. These messages are then transmitted to the NodeGenerator module within this layer. The NodeGenerator module assumes responsibility for processing electronic events received from the Electronics module in Layer 2. It collaborates with the energy management model to calculate the aggregate power consumption of the simulated household. This vital information is then relayed back to the SimModuleCoupling, thereby completing the information exchange loop.

In sum, this multi-level simulation approach, as depicted in the class diagram, forms the bedrock of the smart home scenario. Each layer and its constituent components collaborate seamlessly to replicate the intricate dynamics and interactions within a contemporary smart home environment.

House Security

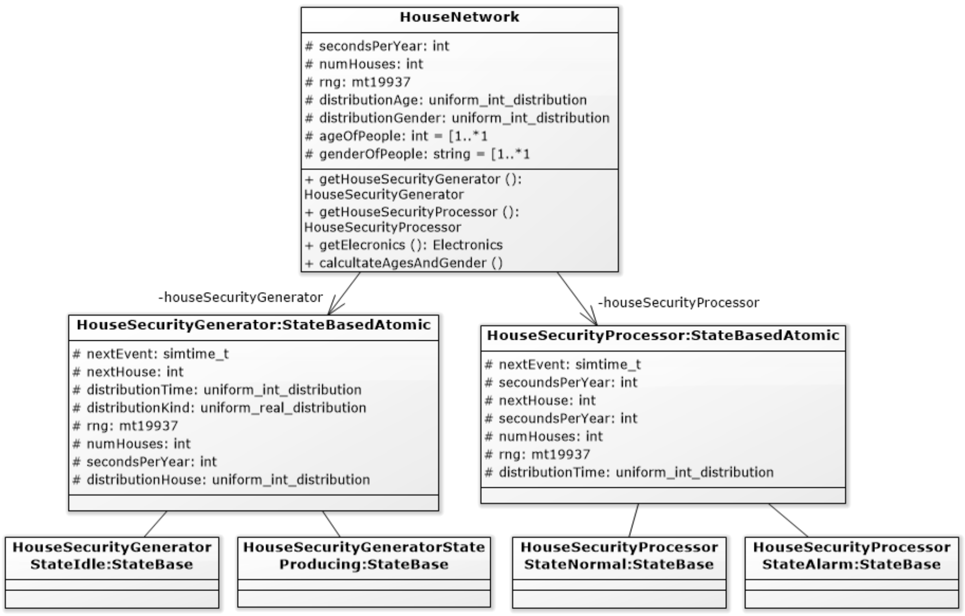

HouseNetwork
# secondsPerYear: int
# numHouses: int
# rng: mt19937
# distributionAge: uniform_int_distribution
# distributionGender: uniform_int_distribution
# ageOfPeople: int = [1..*1
# genderOfPeople: string = [1..*1
+ getHouseSecurityGenerator (): HouseSecurityGenerator
+ getHouseSecurityProcessor (): HouseSecurityProcessor
+ getElecronics (): Electronics
+ calcultateAgesAndGender ()
-houseSecurityGenerator
-houseSecurityProcessor
HouseSecurityGenerator:StateBasedAtomic
# nextEvent: simtime_t
# nextHouse: int
# distributionTime: uniform_int_distribution
# distributionKind: uniform_real_distribution
# rng: mt19937
# numHouses: int
# secondsPerYear: int
# distributionHouse: uniform_int_distribution
HouseSecurityProcessor:StateBasedAtomic
# nextEvent: simtime_t
# secoundsPerYear: int
# nextHouse: int
# secoundsPerYear: int
# numHouses: int
# rng: mt19937
# distributionTime: uniform_int_distribution
HouseSecurityGenerator StateIdle:StateBase
HouseSecurityGeneratorState Producing:StateBase
HouseSecurityProcessor StateNormal:StateBase
HouseSecurityProcessor StateAlarm:StateBase

Figure 5.19.: *Class Diagram of HouseSecurity*

The House Security module plays a pivotal role in simulating security aspects within the smart home environment. It comprises two essential components: the HouseSecurityGenerator and the HouseSecurityProcessor, both based on the StateBasedAtomic approach. Each state has different functions for internal (deltaInternal) and external input (deltaInternal) and timeAdvance, which can be seen in Figure 5.20. This design choice, featuring different states with distinct functions for internal and external inputs, as well as time advancement functions, greatly simplifies the expression of complex behaviors and dependencies, ensuring a comprehensive representation of security dynamics within the simulation.

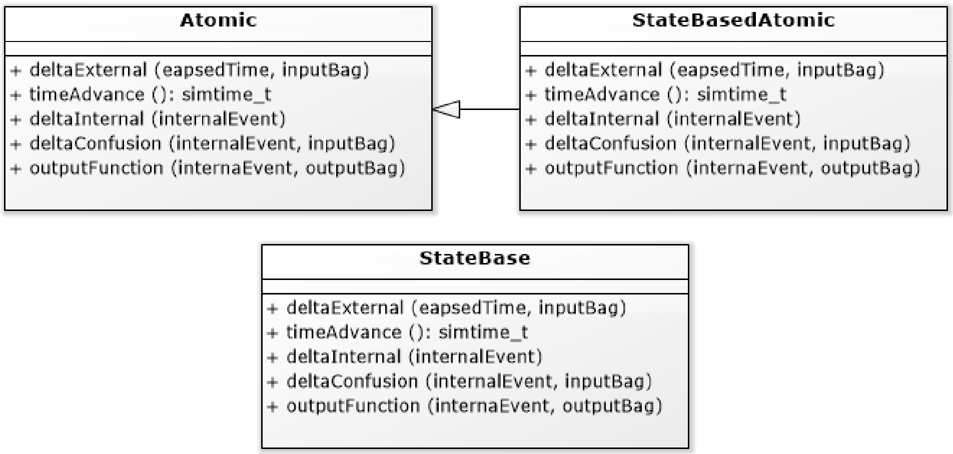

Atomic
+ deltaExternal (eapsedTime, inputBag)
+ timeAdvance (): simtime_t
+ deltaInternal (internalEvent)
+ deltaConfusion (internalEvent, inputBag)
+ outputFunction (internaEvent, outputBag)
StateBasedAtomic
+ deltaExternal (eapsedTime, inputBag)
+ timeAdvance (): simtime_t
+ deltaInternal (internalEvent)
+ deltaConfusion (internalEvent, inputBag)
+ outputFunction (internaEvent, outputBag)
StateBase
+ deltaExternal (eapsedTime, inputBag)
+ timeAdvance (): simtime_t
+ deltaInternal (internalEvent)
+ deltaConfusion (internalEvent, inputBag)
+ outputFunction (internaEvent, outputBag)

Figure 5.20.: *Class Diagram of Atomic and, StateBasedAtomic and StateBase*

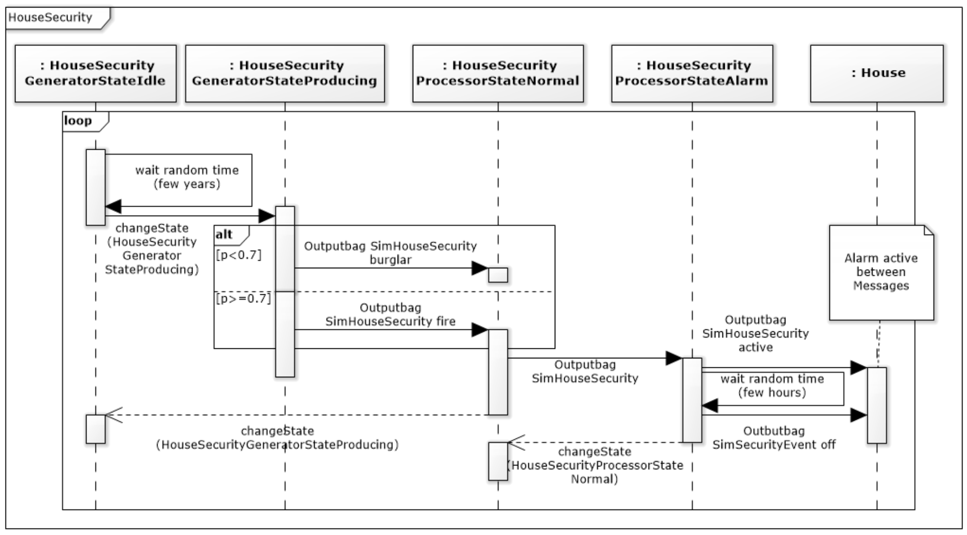

HouseSecurity
: HouseSecurity GeneratorStateIdle
: HouseSecurity GeneratorStateProducing
: HouseSecurity ProcessorStateNormal
: HouseSecurity ProcessorStateAlarm
: House
loop
wait random time (few years)
changeState (HouseSecurity Generator StateProducing)
alt
[p<0.7]
Outputbag SimHouseSecurity burglar
[p>=0.7]
Outputbag SimHouseSecurity fire
Alarm active between Messages
Outputbag SimHouseSecurity active
Outputbag SimHouseSecurity
wait random time (few hours)
changeState (HouseSecurityGeneratorStateProducing)
changeState (HouseSecurityProcessorState Normal)
Outbutbag SimSecurityEvent off

Figure 5.21.: *Sequence diagram of HouseSecurity*

House Security Generator The HouseSecurityGenerator module operates within two primary states: HouseSecurityGeneratorStateIdle and HouseSecurityGeneratorStateProducing.

1. **HouseSecurityGeneratorStateIdle:** This initial state is characterized by a lack of security alarm events. Each house's duration in this state is determined by a predefined timeAdvance function, which results in a random time interval, typically spanning between one day and up to 20 years. During this time, the model remains in a dormant state, awaiting the lapse of the specified duration. This approach introduces variability and a degree of unpredictability to the generation of security events.

2. **HouseSecurityGeneratorStateProducing:** Transitioning from the idle state to this state signifies the initiation of security event generation. Upon entering this state, a security alarm event is generated, simulating security incidents within the household. The nature of the security event is randomly selected, with a 70% likelihood of generating a burglar alarm and a 30% chance of a fire alarm. These probabilities align with real-world statistics, where burglary incidents are more common than fires [98] [99]. The generated security event is placed within the output bag and subsequently transmitted to the HouseSecurityProcessor. Following this action, the model reverts to the HouseSecurityGeneratorStateIdle, resetting the process for the generation of future security events.

House Security Processor The HouseSecurityProcessor module operates in two primary states: HouseSecurityProcessorStateNormal and HouseSecurityProcessorStateAlarm.

1. **HouseSecurityProcessorStateNormal:** In this state, the module remains vigilant for incoming security events. Upon receiving a security event, it is processed and added to the output bag. Subsequently, the event is dispatched via the HouseNetwork to the House, where it triggers the activation of the corresponding alarm within the household. The alarm remains active for a predetermined duration, as specified by the simulation, reflecting real-world alarm systems. Following this activation, the model transitions to the HouseSecurityProcessorStateAlarm.

2. **HouseSecurityProcessorStateAlarm:** In this state, the module orchestrates the duration for which the alarm remains active within the House. Typically, this duration varies randomly between 1 and 24 hours. After this predetermined period, a new security event is generated and sent to the House, effectively simulating the deactivation of the alarm system within the household. The model subsequently returns to the HouseSecurityProcessorStateNormal, ready to process additional security events as they arrive from the HouseSecurityGenerator.

In conclusion, the House Security module effectively replicates the dynamics of security events within the smart home scenario. By incorporating randomness and a state-based approach, it ensures that security incidents are generated and managed in a realistic and dynamic manner, contributing to the overall authenticity of the simulation.

Health Applications

In the context of a smart home, the application of IoT extends beyond mere convenience and energy efficiency. Health monitoring and emergency response systems are pivotal IoT applications designed to enhance the well-being and safety of individuals within the household. These systems incorporate various IoT devices capable of monitoring critical health parameters such as heart rate and respiration rates. Furthermore, these devices are equipped to detect falls or other medical emergencies, triggering automatic alerts for prompt assistance. This aspect is of paramount importance, as a substantial number of accidents and health-related incidents occur within the home environment. One such critical health monitoring device simulated in this use case is a heart rate monitor. The heart rate monitor's primary function is to detect potential coronary events. Simulating the intricacies of a real heartbeat and the detection of a coronary event would be highly complex. Instead, this simulation employs a more straightforward yet effective approach. It considers the age and gender of the individual residing in the house, leveraging a comprehensive study by A. Gößwald, A. Schienkiewitz, E. Nowossadeck, and M. A. Busch [100]. This study, conducted from 2008 to 2011, involved tests, investigations, and surveys on individuals aged between 18 and 79 years. The study's findings, summarized in Table 5.1, provide critical data on the correlation between age, gender, and the likelihood of coronary events. These statistics are utilized within the simulation to determine the probability of a coronary event occurring.

Age	% of women had a coronary	% of men had a coronary
40-49	1.6	3.0
50-59	1.8	6.9
60-69	10.8	19.5
70-79	15.5	30.5

Table 5.1.: *Percentage of people had a coronary at a certain age [100]*

The complete model of the House Health module is depicted in Figure 5.22. This module is constructed as a StateBasedAtomic model encompassing two primary states: HouseHealthGenerator and HouseHealthProcessor.

HouseHealthGenerator

- **HouseHealthGeneratorStateIdle:** This initial state serves as a resting phase in which the module remains inactive, awaiting the initiation of

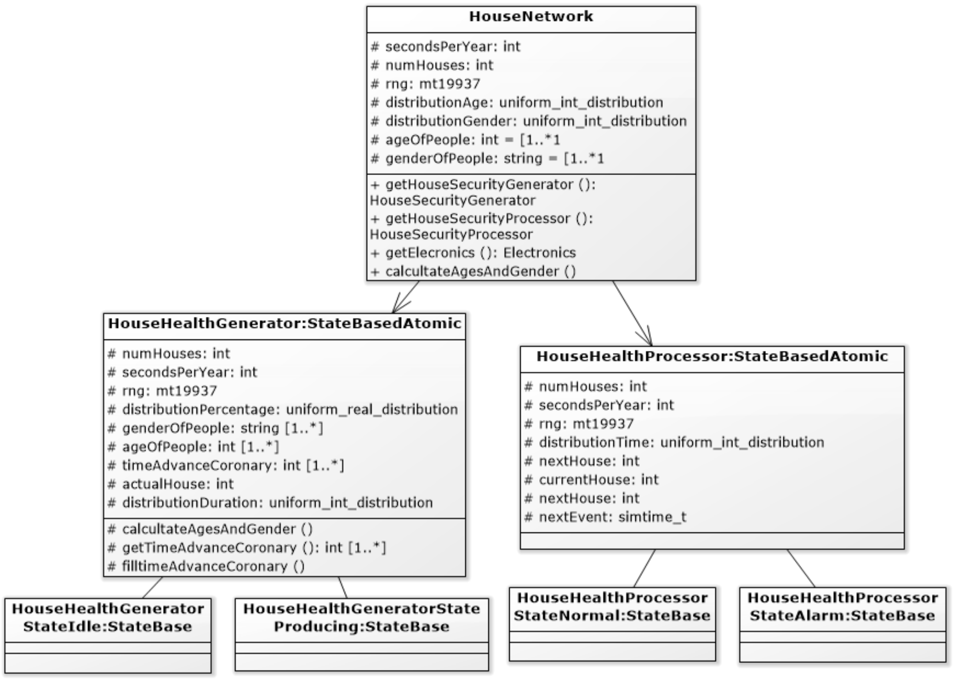

HouseNetwork
# secondsPerYear: int
# numHouses: int
# rng: mt19937
# distributionAge: uniform_int_distribution
# distributionGender: uniform_int_distribution
# ageOfPeople: int = [1..*1
# genderOfPeople: string = [1..*1
+ getHouseSecurityGenerator (): HouseSecurityGenerator
+ getHouseSecurityProcessor (): HouseSecurityProcessor
+ getElecronics (): Electronics
+ calcultateAgesAndGender ()
HouseHealthGenerator:StateBasedAtomic
# numHouses: int
# secondsPerYear: int
# rng: mt19937
# distributionPercentage: uniform_real_distribution
# genderOfPeople: string [1..*]
# ageOfPeople: int [1..*]
# timeAdvanceCoronary: int [1..*]
# actualHouse: int
# distributionDuration: uniform_int_distribution
# calcultateAgesAndGender ()
# getTimeAdvanceCoronary (): int [1..*]
# filltimeAdvanceCoronary ()
HouseHealthProcessor:StateBasedAtomic
# numHouses: int
# secondsPerYear: int
# rng: mt19937
# distributionTime: uniform_int_distribution
# nextHouse: int
# currentHouse: int
# nextHouse: int
# nextEvent: simtime_t
HouseHealthGenerator StateIdle:StateBase
HouseHealthGeneratorState Producing:StateBase
HouseHealthProcessor StateNormal:StateBase
HouseHealthProcessor StateAlarm:StateBase

Figure 5.22.: *Class Diagram of HouseHealth*

health-related events. The duration of this idle state varies for each individual based on the *timeAdvanceCoronary* function, introducing stochasticity and mimicking the unpredictable nature of health events. This approach adds realism to the simulation. During this idle period, the module calculates the number of years until the individual may experience a coronary event. The calculation takes into account the individual's age, gender, and the correlation percentages from the study in table 5.1.

- **HouseHealthGeneratorStateProducing:** Upon transitioning to this state, the module generates a health event corresponding to a potential coronary event. To simulate the stochasticity of health incidents, the simulation employs a calculated time interval for each individual. This interval is based on the calculated years until a potential coronary event. If the time interval expires, an event is dispatched to the House in layer 1, including the houseID. Following this, a new value for timeAdvanceCoronary is computed for the individual. This value is constrained to be between zero and five years since the probability of experiencing a coronary event rises after the first occurrence. Subsequently, the module reverts to the HouseHealthGeneratorStateIdle, where the cycle repeats, ensuring a continuous assessment of coronary event probabilities for individuals.

HouseHealthProcessor The HouseHealthProcessor state-machine model consists of two primary states:

- **HouseHealthProcessorStateNormal:** In this state, the module remains in readiness to receive incoming health events. Upon receiving a health event, it processes the data and adds it to the output bag. Subsequently, the processed event is transmitted via the HouseNetwork to the House, where it influences health-related parameters or triggers an alarm, depending on the nature of the event. The duration of alarm activation, mirroring real-world alarm systems, is controlled by a predetermined time period. Following the expiration of this period, the module transitions to the HouseHealthProcessorStateAlarm.
- **HouseHealthProcessorStateAlarm:** In this state, the module manages the duration for which the health alarm remains active within the House. The duration is typically set to vary randomly between 1 and 24 hours, simulating real-world alarm systems and ensuring the module's actions closely resemble real-life scenarios. After this predetermined period, a new health event is generated and sent to the House, effectively simulating the deactivation of the alarm system within the household. Subsequently, the module transitions back to the HouseHealthProcessorStateNormal, ready to process additional health events as they are received from the HouseHealthGenerator.

In essence, the House Health module simulates health monitoring and emergency response systems within a smart home environment. By considering factors such as age and gender in the context of coronary event probability, it contributes to a more comprehensive and dynamic representation of health-related scenarios within the simulation, enhancing its authenticity and relevance to real-world applications.

Electronic Devices

In the simulated smart home environment, the Electronics component plays a pivotal role in managing electronic devices within the household. This component is structured as a StateBasedAtomic model, encapsulating two primary states: ElectronicsStateIdle and ElectronicsStateProducing (as depicted in Figure 5.23). The Electronics model is responsible for orchestrating the operation of various electronic devices and ensuring their realistic usage patterns, including factors such as whether they are turned on, for how long, and during what times of the day. To realize the diverse array of electronic devices found within a typical household, a fundamental building block called the Device class is employed. Each Device instance is characterized by a unique name and id, serving as identifiers. Additionally, critical device attributes are defined, such as percentageOnPerDay, minTimeOn, maxTimeOn, continuesIfNotAtHome, secondOn, isLightMorning, and isLightEvening, each contributing to the simulation's realism. At the start of each simulated day, the fillElectronicNext() function is invoked (as visualized in Figure 5.24), facilitating the calculation of essential daily parameters. First, the model determines whether the day in question is a working day. Subsequently, it randomly assigns values for standUpTime and sleepTime to emulate the daily routine of

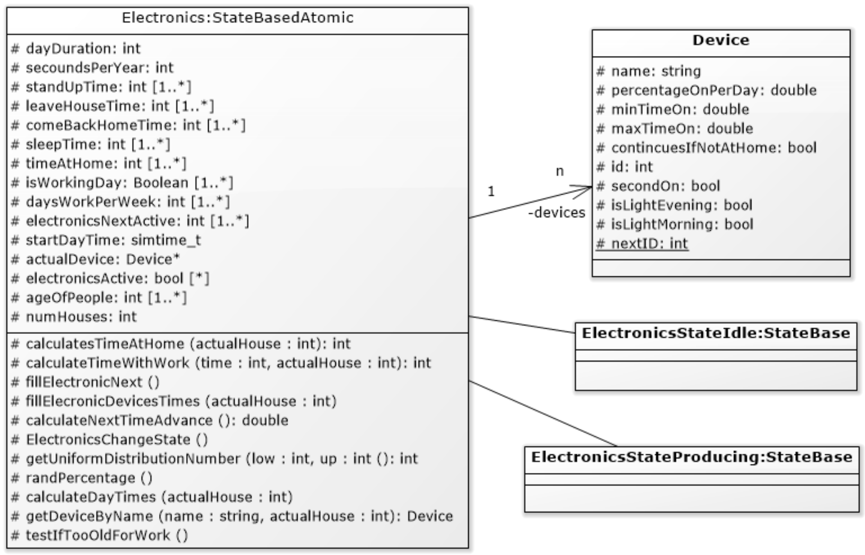

Electronics:StateBasedAtomic
# dayDuration: int
# secoundsPerYear: int
# standUpTime: int [1..*]
# leaveHouseTime: int [1..*]
# comeBackHomeTime: int [1..*]
# sleepTime: int [1..*]
# timeAtHome: int [1..*]
# isWorkingDay: Boolean [1..*]
# daysWorkPerWeek: int [1..*]
# electronicsNextActive: int [1..*]
# startDayTime: simtime_t
# actualDevice: Device*
# electronicsActive: bool [*]
# ageOfPeople: int [1..*]
# numHouses: int
# calculatesTimeAtHome (actualHouse : int): int
# calculateTimeWithWork (time : int, actualHouse : int): int
# fillElectronicNext ()
# fillElecronicDevicesTimes (actualHouse : int)
# calculateNextTimeAdvance (): double
# ElectronicsChangeState ()
# getUniformDistributionNumber (low : int, up : int (): int
# randPercentage ()
# calculateDayTimes (actualHouse : int)
# getDeviceByName (name : string, actualHouse : int): Device
# testIfTooOldForWork ()
1
n
-devices
Device
# name: string
# percentageOnPerDay: double
# minTimeOn: double
# maxTimeOn: double
# contincuesIfNotAtHome: bool
# id: int
# secondOn: bool
# isLightEvening: bool
# isLightMorning: bool
# nextID: int
ElectronicsStateIdle:StateBase
ElectronicsStateProducing:StateBase

Figure 5.23.: *Class Diagram of Electronics*

the household's occupant. In the case of a working day, additional times such as leaveHouseTime and comebackHomeTime are computed using the calculateDayTimes() function. With this information, timeAtHome is determined, representing the duration the occupant spends at home on that particular day.

The fillElectronicDevicesTimes() function is then employed to ascertain whether each electronic device should be activated on that day and, if so, when and for how long it should remain operational. This function optimizes the simulation's realism by considering factors such as the occupant's presence at home and the specific attributes of each device. Before delving into the intricacies of this function, let's first explore the Device class:

- **name** and **id**: Unique identifiers for each electronic device.

- **percentageOnPerDay**: This attribute denotes the likelihood that the device is turned on during a random day. For example, if a device is active every day, its percentageOnPerDay value is set to 1.0.

- **minTimeOn** and **maxTimeOn**: These attributes define the permissible duration for which a device may remain operational.

- **continuesIfNotAtHome**: A boolean attribute that determines whether a device continues to operate even if no one is present at home. For instance, a washing machine may continue to operate irrespective of the occupant's presence, while a radio might be turned off when no one is at home.

- **secondOn**: This attribute is used internally within the Device model and holds no particular significance in the simulation.

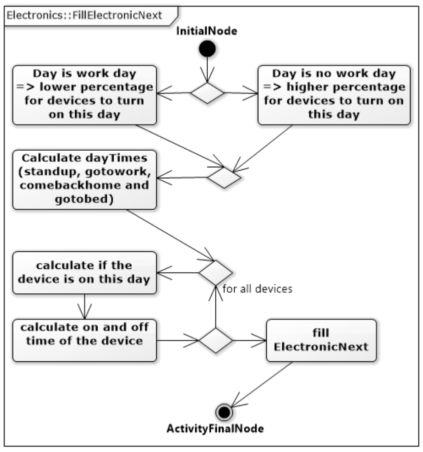

Electronics::FillElectronicNext
InitialNode
Day is work day => lower percentage for devices to turn on this day
Day is no work day => higher percentage for devices to turn on this day
Calculate dayTimes (standup, gotowork, comebackhome and gotobed)
calculate if the device is on this day
for all devices
calculate on and off time of the device
fill ElectronicNext
ActivityFinalNode

Figure 5.24.: *Activity Diagram of the fillElectronicNext function*

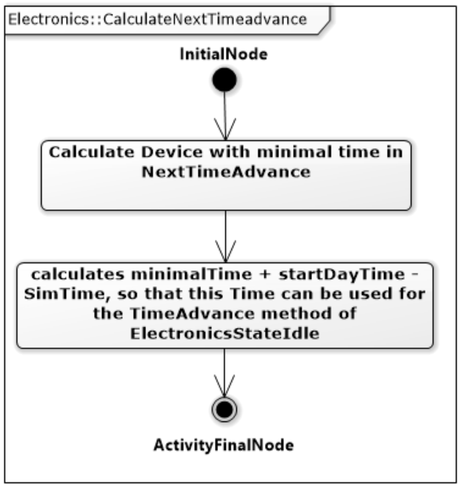

Electronics::CalculateNextTimeadvance
InitialNode
Calculate Device with minimal time in NextTimeAdvance
calculates minimalTime + startDayTime - SimTime, so that this Time can be used for the TimeAdvance method of ElectronicsStateIdle
ActivityFinalNode

Figure 5.25.: *Activity diagram of the calculateNextTimeAdvance function*

- **isLightMorning** and **isLightEvening**: These attributes indicate whether a device is a light source, influencing its behavior in the simulation.

fillElectronicDevicesTimes() This function is instrumental in determining the activation and deactivation times for electronic devices on a given day. Its operation can be broken down into the following steps:

1. **Device Activation Probability**: For each electronic device, the function evaluates whether the device should be turned on on that day based on its percentageOnPerDay attribute. This decision simulates the stochastic nature of device usage.

2. **Duration of Operation**: Once it's established that a device should be active, the function calculates the duration for which the device will remain on. This duration is determined by selecting a random value between minTimeOn and maxTimeOn.

3. **Start Times**: The function calculates the precise start and end times for each device's operation on that day. These times are determined in conjunction with the calculateTimeWithWork() function and are further influenced by random start times. Devices can only be turned on during periods when there is no work or sleep scheduled.

4. **Construction of electronicsNextActive**: The function constructs a list called electronicsNextActive, which contains the exact times at which each device will require activation. Since all devices start in the off state, these times correspond to the initial activation times for each device on that day.

Upon the completion of these steps, the simulation is ready to progress through the day, and the Electronics state transitions to ElectronicsStateNormal. The model remains in this state as long as there are electronic devices requiring activation or deactivation, or until the end of the day is reached. The function calulcateNextTimeAdvance() (as shown in Figure 5.25) facilitates the calculation of the next event times, including both activation and deactivation events, and ensures that the simulation adheres to a realistic timeline. When the ElectronicsStateProducing is entered, the timeAdvance() function is triggered with a value of 0, invoking the outputFunction. In this function, the electronicsStateChange() method is used (as demonstrated in Figure 5.26), enabling the transition of device states. If a device is currently off, its state is changed to active, and the corresponding off time is recorded. Conversely, if a device is already active, it is turned off, and the end-of-day time is assigned since it will remain off for the remainder of the day. The states of all electronic devices are then collected into a list and sent, via the output bag, to the energy management component (as elaborated in section 5.2.3).

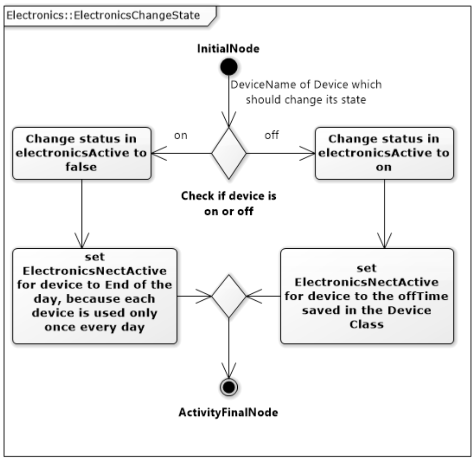

Electronics::ElectronicsChangeState
InitialNode
DeviceName of Device which should change its state
on
off
Change status in electronicsActive to false
Change status in electronicsActive to on
Check if device is on or off
set ElectronicsNectActive for device to End of the day, because each device is used only once every day
set ElectronicsNectActive for device to the offTime saved in the Device Class
ActivityFinalNode

Figure 5.26.: *Activity Diagram of the electronicsStateChange function*

Upon reaching the end of the day, the fillElectronicNext() function is invoked again, the state transitions back to ElectronicsStateIdle, and a new day's simulation begins. This iterative process continues, faithfully emulating the operation of electronic devices within the smart home environment and adhering to realistic usage patterns influenced by the occupant's daily routine. The Electronics is also a StateBasedAtomic model. It has the main class Electronics and the states ElectronicsStateIdle and ElectronicsStateProducing, which can be seen in Figure 5.23. To realize the different electronic devices, a class called Device is used. In this model, all important things of a day including the sleep and the work times of the person living in this house are managed. Also the ontimes of the devices are calculated. A device can only be turned on, if the person is at home. If the person does not work a day, more electronic devices are used this day. At the start of each day, the function fillElectronicNext() (see 5.24) is called, where the basic day data are calculated.

First it is decided, if the day is a working day or not. After that the standUp-Time and sleepTime of the person living in the house is randomly determined. If it is a working day, also the leaveHouseTime and the comebackHomeTime are calculated with the calculateDayTimes() function. With this information the calculateTimeAtHome() method can determine the timeAtHome. Now the fillElectronicDevicesTimes() function, which decides if and when a Device is turned on this day, is called. Before explaining this function the Device class must be presented. In the Device class all important information about a Device is stored. Its name and id are the unique identifiers of the device. PercentageOnPerDay means the chance the device is on at a random day. For example if the light is on every day, so its PercentageOnPerDay will be 1.0. If the device is on, it is on between minTimeOn and maxTimeOn. For devices which continue to be on, also if no one is at home, there is continuesIfNotAtH-ome. This means this attribute is true, if for example the device is a washing machine, because it will be on, even if the person goes to work or bed. If the device for example is a radio, the device will be turned off, if there is no one at home. SecondOn is only important for the implementation of Device and has no significant meaning. The last two attributes are isLightMorining and isLightEvening. They indicate, if the device is a light. Now the fillElec-tronicDevicesTimes() can be explained. With the help of this function the on and off times of the devices are calculated. First of all for each device is determined with percentageOnPerDay, whether the device is on or not this day. After that is calculated, how long the device should be on this day. For this purpose, a random number between minTimeOn and maxTimeOn is used. Afterwards with this and the help of the calculateTimeWithWork() method and a random start time (only times with no work or sleep are possible) the exact on and off times of each device on this day are calculated. Once all this is done, the electronicsNextActive is filled. In this list, the exact times for the next events of each device are stored. In this case these are the on times of the devices, because all devices are off. After all this is done, the simulation of the day can start and the state is changed to ElectronicsStateNormal. The model stays as long as there is no electronic device which needs to be turned on or off or the day ends. This is calculated with the calulcateNextTimeAdvance() function, which can be seen in Figure 5.25. This method is located in the Electronics class, iterates through the nextTimeAdvance list and returns the minimum value, which is used for the timeAdvance() function of the current state. Also the device with this time is stored in actualDevice. Afterwards the state is changed to ElectronicsStateProducing. The timeAdvance() of this state is 0, so that the outputFuction is called. In this method the electron-icsStateChange() function in Electronics is used first, where the next times are filled in nextTimeAdvance. (refer to Figure 5.26) If the device is off, its state changes to active and the off time is filled into it. If the device is on, it is turned off and the end of day time is used, because the device will be off for the rest of the day. Now the states of all electronic devices are stored in a list and sent with the output bag to the energy management tool. This is explained in section 5.2.3. If the end of the day is reached, the fillElectronicNext() function is called again and after the state is changed back to ElectronicsStateIdle a new day can start. If it is not the end of the day, this is done too.

House Energy Calculation

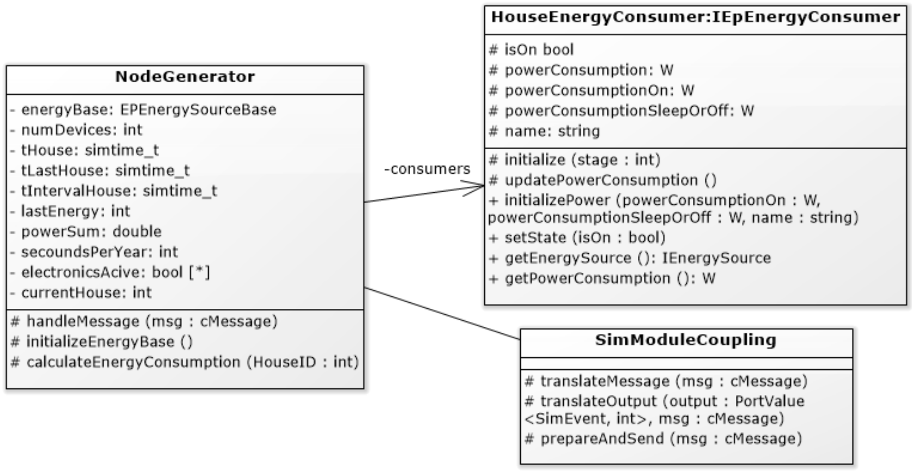

NodeGenerator
- energyBase: EPEnergySourceBase
- numDevices: int
- tHouse: simtime_t
- tLastHouse: simtime_t
- tIntervalHouse: simtime_t
- lastEnergy: int
- powerSum: double
- secoundsPerYear: int
- electronicsAcive: bool [*]
- currentHouse: int
# handleMessage (msg : cMessage)
# initializeEnergyBase ()
# calculateEnergyConsumption (HouseID : int)
-consumers
HouseEnergyConsumer:IEpEnergyConsumer
# isOn bool
# powerConsumption: W
# powerConsumptionOn: W
# powerConsumptionSleepOrOff: W
# name: string
# initialize (stage : int)
# updatePowerConsumption ()
+ initializePower (powerConsumptionOn : W, powerConsumptionSleepOrOff : W, name : string)
+ setState (isOn : bool)
+ getEnergySource (): IEnergySource
+ getPowerConsumption (): W
SimModuleCoupling
# translateMessage (msg : cMessage)
# translateOutput (output : PortValue <SimEvent, int>, msg : cMessage)
# prepareAndSend (msg : cMessage)

Figure 5.27.: *Class Diagram of INET*

The energy calculation within the smart home scenario is a crucial aspect that occurs at layer 3 in the OMNeT++ framework, facilitated by the INET framework (as illustrated in Figure 5.27). In Figure 5.28, a sequence diagram depicts how the status of electronic devices traverses this segment of the simulation. The process begins with the output bag generated by the Electronics model, which is then relayed to the HouseCoordinator.

HouseCoordinator This intermediary class plays a vital role in managing the flow of data within the simulation. It takes the output bag from the Electronics model and, through interaction with the core simulator, forwards it to the SimModuleCoupling module in the INET framework.

SimModuleCoupling Located within the INET framework, this module serves as an interface between the OMNeT++ simulation and the INET simulation. Its primary function is to translate and transmit data between these two frameworks. In the context of energy management, it leverages the translateOutput function to convert the output bag from OMNeT++ to an appropriate format for the INET part, encapsulated within a cMessage. This message is then dispatched to the NodeGenerator within the INET framework.

NodeGenerator Within the INET framework, the NodeGenerator module is responsible for handling various aspects of simulation coordination and execution. In the context of energy management, it receives the message containing the status of electronic devices from SimModuleCoupling via the handleMessage method. Subsequently, the NodeGenerator undertakes the critical task

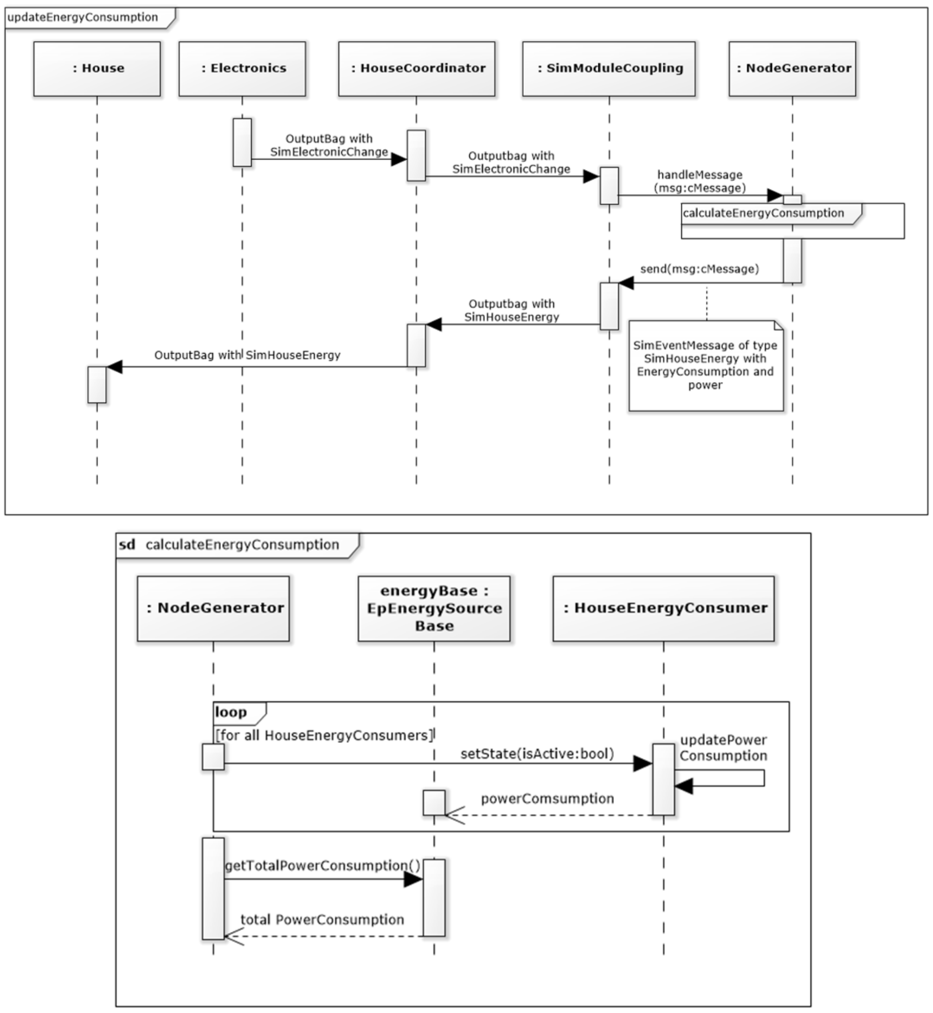

updateEnergyConsumption
: House
: Electronics
: HouseCoordinator
: SimModuleCoupling
: NodeGenerator
OutputBag with SimElectronicChange
Outputbag with SimElectronicChange
handleMessage (msg:cMessage)
calculateEnergyConsumption
send(msg:cMessage)
Outputbag with SimHouseEnergy
OutputBag with SimHouseEnergy
SimEventMessage of type SimHouseEnergy with EnergyConsumption and power
sd calculateEnergyConsumption
: NodeGenerator
energyBase : EpEnergySource Base
: HouseEnergyConsumer
loop
[for all HouseEnergyConsumers]
setState(isActive:bool)
updatePower Consumption
powerComsumption
getTotalPowerConsumption()
total PowerConsumption

Figure 5.28.: *Sequence diagram showing the energy consumption flow*

of calculating the energy consumption within the smart home, a process explained in detail below.

Energy Consumption Calculation Before energy consumption can be computed, the energy management tool within the NodeGenerator must be appropriately initialized. The initializeEnergyBase function fulfills this role. In this method, key attributes of each electronic device are set, encompassing their energy consumption during both the active (on) and standby (off) states. Notably, some devices continue to draw substantial power even when switched off, as is often the case with modern appliances. Subsequently, an energyBase is instantiated, representing the foundation upon which energy calculations are executed. Within the energyBase, each device is incorporated as an energyConsumer, a core concept in energy management. A novel type of energy consumer, HouseEnergyConsumer, is introduced, extending the IEpEnergyConsumer interface from the INET framework. This interface accommodates two distinct power consumption values: powerConsumptionOn and powerConsumptionOff, representing the power consumed when a device is active and in standby, respectively. The actual power consumption value, powerConsumption, dynamically adjusts based on the state of the device. The setState(isOn: bool) function governs this transition, enabling the activation or deactivation of devices as they operate within the smart home. To retrieve the current power consumption, the getPowerConsumption() method is employed.

Message Handling The handleMessage(msg: cMessage) method within the NodeGenerator is responsible for receiving and processing messages, primarily those transmitted from SimModuleCoupling. When a message of type SimElectronicChange is received, this method triggers the execution of the calculateEnergyConsumption() function. The primary purpose of this function is to compute the total energy consumption of the smart home and subsequently transmit this data back to the OMNeT++ portion of the simulation.

The energy calculation process unfolds as follows:

1. **Updating Energy Consumers**: The calculateEnergyConsumption() function commences by updating the status of each HouseEnergyConsumer, aligning their states with the information received from the SimElectronicChange message. This step ensures that the simulation accurately reflects the current operational state of each electronic device.

2. **Total Energy Calculation**: With the states of all HouseEnergyConsumers appropriately configured, the function proceeds to calculate the total energy consumption. This calculation leverages the energyBase, which encompasses all registered energy consumers, both electronic devices and standby power sources. Additionally, the power consumed throughout the entire day (totalPower) is determined.

3. **Data Transmission**: Following the computation of total energy consumption and daily power consumption, these values are encapsulated

within a new cMessage. This message is then dispatched back to the SimModuleCoupling module within the OMNeT++ framework, facilitating the transfer of energy consumption data from the INET framework to the OMNeT++ simulation.

By effectively coordinating this energy calculation process between OMNeT++ and the INET framework, the smart home simulation maintains an accurate representation of energy consumption. This information is vital for assessing the efficiency and sustainability of the smart home environment and its electronic devices.

5.3. Knowledge Base in SysMD

In this section, we translate the use case into the SysMD language. The use case is reconstructed using SysMD Notebook to showcase its capabilities in a real-world context. Please note that the knowledge base presented in this chapter has been previously disseminated in [101], and has been thoughtfully incorporated into this dissertation to provide a comprehensive and coherent perspective on the subject matter.

5.3.1. Overview

The model comprises two fundamental components: the knowledge base and the explicit scenarios under examination. The knowledge base is a repository of essential facts, information, legislative aspects, and standards. It serves as the foundation for common knowledge, preventing redundancy in modeling, fostering knowledge sharing across diverse domains, and promoting knowledge reuse. Additionally, the knowledge base aids in establishing a fundamental model structure, including element classes and their associated properties. The scenarios section delves into concrete models and situations. Here, we leverage the foundational structure and information from the knowledge base, enriching it with specific data and assessing its compliance with the knowledge base. This approach essentially constructs a solution space for the given modeling problem. Now, let's delve into the specific example: modeling a SmartGrid consisting of multiple houses with consumers and their electricity demand. We'll explore this example from a bottom-up perspective. At the lowest layer, we encounter various consuming appliances. Moving up a level, we find different houses, each equipped with a range of appliances. These houses are then connected to an integrated power supplier. Across all these levels, we can calculate various forms of power demands, such as standby or full usage, and subsequently verify if they align with the power supply specifications. The model can be adjusted and expanded in numerous ways to explore different calculation outcomes. Within this environment, we present scenarios that represent real-world assumptions for modeling purposes. These concrete designs and their corresponding specifications are detailed in the subsequent sections. This comprehensive approach highlights the versatility of SysMD Notebook in tackling complex, real-world scenarios.

5.3.2. Smart Power Knowledge Base

In this initial subsection, we establish the foundation of our model, which serves as the knowledge base for our SmartGrid environment. Within this knowledge base, we create the SmartGrid package to encapsulate all the high-level structures, classes, and relationships, laying the groundwork for subsequent detailed models. Figure 5.29 provides a visual representation of this knowledge base. Our approach begins by defining a set of components that are fundamental to our models. We then proceed to assign properties to these previously defined components. These properties include essential attributes such as a name, data type (e.g., String, Real, or Boolean), a value or value range, and optionally, a physical unit if required for precise modeling. Furthermore, we identify and define potential appliances that may be necessary in more refined models. A notable feature of these definitions is the implicit use of inheritance. Specifically, in lines one to six, we define the Houseconsumer component along with all its associated properties. By specifying that an appliance "isA" Houseconsumer, we implicitly inherit all the properties of the Houseconsumer component. This approach minimizes redundancy and the potential for errors that could arise from rewriting properties. This knowledge base provides a structured and efficient way to establish the core elements and their characteristics, setting the stage for the subsequent development of more intricate SmartGrid models.

5.3.3. Scenarios in SysMD

In this section, we delve into various scenarios modeled using SysMD, with a focus on a real-world SmartGrid environment. These scenarios demonstrate the versatility and practicality of SysMD in capturing complex systems.

Scenario 1: Multi-House SmartGrid

Figure 5.30 illustrates the first scenario, which depicts a SmartGrid comprising seven houses, each equipped with different electric appliances. Leveraging the knowledge base established earlier, we reuse pre-defined models, enhancing modeling efficiency. Initially, we designate each house as a microgrid, enabling them to inherit properties defined in the knowledge base.

Additionally, we specify certain house appliances as specializations of Houseconsumers. Subsequently, we assign values to the properties of these appliances, ensuring they fall within the range specified by their parent class, thereby maintaining consistency.

For each house, we further define the assignment of various appliances, allowing for diverse configurations. This flexibility enables us to assign appliances multiple times or alter their orders. Under the heading of Gridsupply, we define an electricity supplier to which the houses are connected. The properties defined for the supplier derive their values by aggregating the respective values of the connected houses. This allows for the comparison of these values

```
Document uses ISO26262.
Document imports ISO26262, ScalarValues.

Global hasA Package Smartgrid.
Smartgrid defines
  microgrid isA Component;
  AASsystem isA Component;
  EhealthAsystem isA Component;
  Nodegenerator isA Component;
  Houseconsumer isA Component;
  EnergyMobilitysystem isA Component.

Smartgrid::Houseconsumer hasA
  Value isOn: Boolean = true,
  Value powConsumption: Real(1..100) [W],
  Value powConsumptionOn: Real(1 .. 10000) [W],
  Value powConsumptionSleepOff: Real(1..100) [W],
  Value name: String.

Smartgrid::microgrid hasA
  Value powConsumptionOn: Real [W] =
    sumOverParts(powConsumptionOn),
  Value powConsumption: Real [W] = sumOverParts(powConsumption),
  Value powConsumptionSleepOff: Real [W] =
    sumOverParts(powConsumptionSleepOff).

Smartgrid defines
  Smokedetec isA Houseconsumer;
  Electronics isA Houseconsumer;
  ColorLamp isA Houseconsumer;
  Motiondetec isA Houseconsumer;
  Smokedetec isA Houseconsumer;
  Soundddetec isA Houseconsumer;
  CVcharging isA Houseconsumer;
  BiometricAccess isA Houseconsumer;
  HVAC isA Houseconsumer;
  Falldedtect isA Houseconsumer;
  HeartRatemonitor isA Houseconsumer;
  Respirationmonitor isA Houseconsumer;
  Energystorage isA Houseconsumer;
  Smartplug isA Houseconsumer;
  Watermeter isA Houseconsumer;
  Electronics isA Houseconsumer;
  Lightening isA Houseconsumer.
```

Figure 5.29.: *SmartGrid Knowledge Base*

```
Microgrid hasA Package Consumers.
Microgrid defines
    gridSupply isA Component;
    gridParking isA Component;
    house1 isA Smartgrid::microgrid;
    house7 isA Smartgrid::microgrid;
    house6 isA Smartgrid::microgrid;
    house5 isA Smartgrid::microgrid;
    house4 isA Smartgrid::microgrid;
    house3 isA Smartgrid::microgrid;
    house2 isA Smartgrid::microgrid.

Microgrid::Consumers defines
    Fridge isA Smartgrid::Electronics;
    Smoke isA Smartgrid::Smokedetec;
    Doorcam isA Smartgrid::BiometricAccess;
    Smartlights isA Smartgrid::Lightening;
    HVAC isA Smartgrid::HVAC;
    Washer isA Smartgrid::Electronics.
```

Figure 5.30.: *SmartGrid Scenario 1*

```
Smoke hasA
    Value powConsumption: Real(2..50) [W],
    Value powConsumptionOn: Real(70..99) [W],
    Value powConsumptionSleepOff: Real(1..5) [W].

HVAC hasA
    Value powConsumption: Real(2..50) [W],
    Value powConsumptionOn: Real(1000..3000) [W],
    Value powConsumptionSleepOff: Real(10..20) [W].
```

Figure 5.31.: *Properties per Consumer*

```
house1 hasA
    Part Smoke: Consumers::Smoke,
    Part HVAC: Consumers::HVAC,
    Part Fridge: Consumers::Fridge,
    Part Doorcam: Consumers::Doorcam.

house2 hasA
    Part Smoke: Consumers::Smoke,
    Part HVAC: Consumers::HVAC,
    Part Smartlights: Consumers::Smartlights,
    Part Doorcam: Consumers::Doorcam.
```

Figure 5.32.: *Here we assign different consumers to the different generic houses and can thereby check their respective single consumptions.*

with the supplier's defined maximum power supply, facilitating a comprehensive assessment.

```
gridSupply hasA
    Part microconsumer1: house1,
    Part microconsumer2: house2,
    Part microconsumer3: house3,
    Part microconsumer4: house4,
    Part microconsumer5: house5,
    Part microconsumer6: house6,
    Part microconsumer7: house7,
    Value isOn: Boolean(true),
    Value powConsumption: Real [W] =
      sumOverParts(powConsumption),
    Value powConsumptionOn: Real [W] =
      ITE(isOn, sumOverParts(powConsumptionOn),0.0),
    Value powConsumptionSleepOff: Real [W] =
      sumOverParts(powConsumptionSleepOff),
    Value maxPowpeak: Real(8..8) [kW],
    Value b: Boolean = powConsumptionOn <= maxPowpeak.
```

Figure 5.33.: *The connection of the different houses to one supplier grid.*

Scenario 2: Integration of Microgrids

The second scenario, depicted in Figure 5.34, aims to integrate the previous scenarios into a multi-dependent SmartGrid. Here, various microgrids, such as houses and a smart parking facility, are modeled to be part of a comprehensive SmartGrid system. We explore the connections between consuming entities (e.g., electric appliances and car chargers), power supplies, and a battery with specific capacity.

```
Microgrid defines
    battery isA Component;
    connectGrid isA Component.

Microgrid::battery hasA
    Value capacity: Real(1200..1200) [kWh].

Microgrid::connectGrid hasA
    Part battery: Microgrid::battery,
    Part housegrid: Microgrid::gridSupply,
    Value mainsupplyOn: Boolean(true),
    Value mainsupply: Real(2..2) [MW],
    Value powDemand: Real [kW] = sumOverParts(powConsumption),
    Value outageCover: Boolean =
      ITE(mainsupplyOn, powDemand < mainsupply,
          powDemand < (battery::capacity / 0.5[h])).
```

Figure 5.34.: *In the second scenario, we connect the two power supplies with a battery as a backup power source. We assume a maximum outage of 30 minutes.*

To account for power outage scenarios, we introduce a simple Boolean model indicating the presence of an outage. However, this model can be expanded with more complex time-based representations. This comprehensive approach allows us to model uncertainties in demand and supply, representing complex information across multiple levels and propagating calculations from the knowledge base up to the meta model. In doing so, we create a powerful means of representing intricate systems in a human- and computer-readable format, adhering to standard modeling languages.

5.4. Implementation of the performance Evaluation

This section elucidates the software implementation process of the model, as expounded in Section 5.2.3. The realization of this model is rooted in the utilization of the OMNeT++ simulation framework [95], specifically version 5.4.1, with C++ as the underlying programming language. The necessity of adhering to a minimum C++14 standard arises as any divergence from this specification impedes the software's compilation process. It is imperative to note that prior to engaging with the software, the installation of requisite packages, as stipulated in the OMNeT++ user manual, is mandatory. Upon fulfilling these prerequisites, the OMNeT++ environment can be initiated, the use case can be seamlessly imported. Furthermore, it is essential to incorporate the INET framework [1], version 4.0, within the same project, as the newer

[1]https://inet.omnetpp.org

iterations are incompatible with OMNeT++ 5.4.1. Subsequent to the successful compilation of the software, the implementation phase can commence.

```
HouseNetwork::HouseNetwork()
{
  // Basic house data
  secondsPerYear = 365 * 24 * 60; // Total seconds in a year
  numHouses = 10; // Number of houses

  // Calculate ages and gender distribution
  calculateAgesAndGender();

  // Initialization
  coordinator = new HouseCoordinator();
  houseSecurityGenerator = new HouseSecurityGenerator
    (100, secondsPerYear, numHouses); // 100 = SecurityInterval
  houseSecurityProcessor = new HouseSecurityProcessor
    (secondsPerYear, numHouses);
  houseHealthGenerator = new HouseHealthGenerator
    (&ageOfPeople, &genderOfPeople, secondsPerYear, numHouses);
  houseHealthProcessor = new HouseHealthProcessor
    (secondsPerYear, numHouses);
  electronics = new Electronics(&ageOfPeople, secondsPerYear, numHouses);

  // Seed for randomization
  seed = std::chrono::system_clock::now().time_since_epoch().count();

  // Establish connections
  connect(houseSecurityGenerator, 0, houseSecurityProcessor, 0);
  connect(houseHealthGenerator, 0, houseHealthProcessor, 0);
  // ... Additional connections
}
```

Figure 5.35.: *Constructor of the HouseNetwork implementation*

Within the HouseNetwork constructor (refer to Figure 5.35), several fundamental aspects of the model are addressed. This includes the definition of key parameters, such as the number of houses and the duration of a year in seconds. Additionally, the ages and gender distribution of the residents are computed. The subsequent steps encompass the instantiation of various model components:

- **HouseCoordinator**: A central coordinating entity.
- **HouseSecurityGenerator**: Responsible for generating security events.
- **HouseSecurityProcessor**: Processes security events.
- **HouseHealthGenerator**: Generates health events based on age and gender.
- **HouseHealthProcessor**: Analyzes health events.

- **Electronics**: Manages electronic devices within the houses.

Furthermore, the seeding of a random number generator is performed to introduce stochastic elements into the simulation. Finally, connections between these components are established to facilitate the flow of information within the simulation. The above code snippet embodies the foundational setup for the ensuing simulation within the OMNeT++ framework.

5.4.1. HouseNetwork and connection of the modules

This section expounds upon the establishment of the HouseNetwork and the interconnection of modules within the presented showcase model, in accordance with the framework delineated in Section 5.2.3. Within the HouseNetwork constructor, a multitude of crucial parameters and components are defined and initialized (refer to Figure 5.35). Notably, the attributes secondsPerYear' and numHouses' are set, with secondsPerYear' serving as a variable that can adjust the simulation's speed. Setting secondsPerYear' to 2436560, equating a simulated second to a real-time minute, helps prevent excessive loss of simulation detail, a concern when using smaller values. 'numHouses' is a defining parameter, governing the number of simulated houses. In the main method (refer to Figure 5.36), the initialization of the house simulation is orchestrated. Firstly, a new HouseNetwork and a House, intended for connection to the network module, are instantiated. Subsequently, an outputBag is initialized, designed to establish connections among all house modules within Layer 2. With the aid of port connections in the HouseNetwork, this process is streamlined, ensuring each module receives only the requisite input. A new HierarchicalHouse, which binds the House to the HouseNetwork through Layer 2's output bag, is generated. Additionally, the House is linked to the layer1scenario since it resides within this layer. To facilitate communication between the main simulation and the OMNeT++ simulation, each module in Layer 2 is incorporated into the showcaseCouplingTable, with corresponding simulator entries in the showcaseCoupledSimulator table. These tables establish the essential links enabling message exchanges between the two simulation environments.

The HouseCoordinator class (refer to Figure 5.37) plays a pivotal role in orchestrating communication between the primary simulation and the OMNeT++ simulation. The deltaExternal method iterates through the inputBag, forwarding each event via the outputFunction to either OMNeT++ or the HouseNetwork, thus enabling bidirectional communication. This communication is facilitated through a shared Environment, as described in Section 4.5.1.

5.4.2. House Security

The House Security model, a pivotal component within the use case, is composed of two primary constituents: the HouseSecurityGenerator and the HouseSecurityProcessor, both possessing distinct states. These modules are intricately linked within the broader HouseNetwork. The HouseSecurityGenera-

```
HouseNetwork* layer2Network_house = new HouseNetwork();
House* layer2house = layer1scenario->getHouse();
auto l2OutputBag_house = Bag<Atomic<PortValue<SimEvent, int>,
  simtime_t>*>();
layer2OutputBag_house.insert(..(l2Network_house->getElecronics()));
layer2OutputBag_house.insert(..(l2Network_house->getCoordinator()));
[...] //insert other modules
HierarchicalHouse* hierarchicalHouse = new HierarchicalHouse
(layer2house,layer2OutputBag_house,layer2Network_house);
layer1scenario->setHouse(hierarchicalHouse);
showcaseCoupledSimulators[2] = hierarchicalHouse->getSimulator();
showcaseCouplingTable[2] = layer2Network_house ->getCoordinator();
[...] // Other entires in the coupling tables
```

Figure 5.36.: *House part of the main function of the simulation*

```
void HouseCoordinator::deltaExternal([...] elapsedTime, [...] inputBag)
{
    for (auto iter = inputBag.begin(); iter != inputBag.end(); iter++)
    {
        int port = (*iter).port;
        auto type = (*iter).value.getType();
        if (port == electronicsPort){
            PortValue<SimEvent, int> input = (*iter);
            omnetInput.push_back(input);
        }
       [...] //other SimEvent types
    }
}
// Adds networkInput and omnetInput to outputBag
void HouseCoordinator::outputFunction([...] internalEvent, [...] outputBag)
{
    if (!omnetInput.empty()){
        for (auto iter = omnetInput.begin(); iter != omnetInput.end(); iter++){
            auto output = (*iter);
            outputBag.insert(output);
        }
        [...] // the same for the networkInput
    }
}
//if network or omnet input is empty
simtime_t HouseCoordinator::timeAdvance()
{
    if ((!networkInput.empty()) || (!omnetInput.empty()))
        return pdevs::pdevsZero<simtime_t>();
    return pdevs::pdevsInf<simtime_t>();
}
```

Figure 5.37.: *Parts of the HouseCoordinator*

tor encompasses the main class along with two states, HouseSecurityGeneratorStateIdle and HouseSecurityGeneratorStateProducing. The constructor for this model, as portrayed in figure 5.38, assumes a pivotal role in parameterization. It establishes basic values and defines key distributions. Notably, the distributionTime is instrumental in generating a HouseSecurity event, introducing a randomized waiting period ranging from one day to several years. The randInterval parameter signifies the average time span between security events in years. As the number of houses increases, the necessity for producing more security events amplifies. Consequently, the maximum waiting time for an event, calculated as $randInterval * 2 * secondsPerYear$, is divided by the number of houses to ascertain the maximum value for the distribution. The distributionKind is responsible for computing a random percentage, subsequently utilized in determining the nature of the impending event. It should be noted that in approximately 70% of instances, the event is categorized as a fire, with the remaining cases designated as burglaries.

```
HouseSecurityGenerator(int randInterval,int secondsperyear,int numHouses){
    auto seed = chrono::system_clock::now().time_since_epoch().count();
    rng = new mt19937(seed);
    processorPort = 0;
    this−>numHouses=numHouses;
    this−>secondsperyear=secondsperyear;
    distributionTime= uniform_int_distribution<int>((int)(secondsperyear/(365)),
        (int)((randInterval*2*secondsperyear)/numHouses));
    distributionKind = std::uniform_real_distribution<double>(0, 1);
    distributionHouse = uniform_int_distribution<int>(1,numHouses);
    nextEvent = distributionTime(*rng);
    nextHouse = distributionHouse(*rng);
    this−>currentState = HouseSecurityGeneratorStateIdle::getInstance();
}
```

Figure 5.38.: *Constructor of HouseSecurityGenerator*

Following the random number calculations, the state transitions to HouseSecurityGeneratorStateIdle. Subsequently, the model awaits the calculated time interval through the timeAdvance function. Once this interval elapses, the state transitions to HouseSecurityGeneratorStateProducing. Within this state, the outputFunction (see Figure 5.39) takes center stage. It employs the distributionKind to compute a random percentage and stores it as kind. This percentage determines the nature of the event, with approximately 70% likelihood of it being a burglary, and the remaining 30% leading to a fire. The houseID for the event is randomly selected and paired with the event type, encapsulated as a SimEvent, which is then inserted into the outputBag. This outputBag subsequently forwards the event to the HouseSecurityProcessor.

The HouseSecurityProcessor receives events from the HouseSecurityGenerator and awaits a random duration, ranging from 1 to 24 hours. After this interval, a security event is dispatched to signal the House to deactivate the alarm, as demonstrated in figure 5.40.

```
void HouseSecurityGeneratorStateProducing::outputFunction(ctx,internalEvent,outputBag)
{
    auto gen = static_cast<HouseSecurityGenerator*>(ctx);
    double kind= gen->distributionKind(*(gen->rng));
    SimHouseSecurity securityEvent;
    if(kind<=0.7){
        securityEvent.fireDetencted=false;
        securityEvent.burglarDetected=true;
    }
    else if(kind<=1.0){
        securityEvent.fireDetencted=true;
        securityEvent.burglarDetected=false;
    }
    securityEvent.houseID=gen->nextHouse;
    SimEvent event(securityEvent);
    PortValue<SimEvent, int> output(gen->processorPort, event);
    outputBag.insert(output);
}
```

Figure 5.39.: *outputFunction of HouseSecurityGeneratorStateProducing*

```
void HouseSecurityProcessorStateAlarm::outputFunction( ctx, internalEvent, outputBag)
{
    SimHouseSecurity securityEvent;
    securityEvent.fireDetencted=false;
    securityEvent.burglarDetected=false;
    auto processor = static_cast<HouseSecurityProcessor*>(ctx);
    securityEvent.houseID=processor->nextHouse;
    SimEvent event(securityEvent);
    PortValue<SimEvent, int> output(0, event);
    outputBag.insert(output);
}
```

Figure 5.40.: *outputFunction of HouseSecurityProcessorStateAlarm*

The HouseSecurityProcessor module, during its HouseSecurityProcessorStateAlarm state, ensures that an appropriate signal is generated to deactivate the alarm system in the House module. It is essential to note that the generated signal is devoid of event-specific information, setting the fireDetected and burglarDetected flags to false. This ensures that the alarm is consistently deactivated for all event types. The houseID is incorporated into the signal, allowing the House module to identify the specific house that should deactivate its alarm system. In summary, the House Security model employs a systematic approach to generating security events, effectively simulating both fire and burglary events with stochasticity, and coordinates the activation and deactivation of alarms within the framework. The seamless interaction between the HouseSecurityGenerator and HouseSecurityProcessor modules, coupled with randomized event generation, enhances the realism and versatility of the simulation.

5.4.3. House Health Services

In accordance with the concepts outlined in Section 5.2.3, the implementation of the House Health simulation model consists of two core components: the HouseHealthGenerator and the HouseHealthProcessor. Here, we delve into the technical details of these components and the processes that underpin their operation.

HouseHealthGenerator The central functionality of the HouseHealthGenerator is embodied in the calculateCoronary method, as depicted in figure 5.41. At its core, this method determines the probability of a coronary event occurring based on factors such as gender and age. This probability is then utilized to ascertain the age at which a person living in the house is likely to experience a coronary event, drawing from statistical data represented in Table 5.1.

The state of the HouseHealthGenerator is initially set to HouseHealthGeneratorStateIdle. It remains in this state until the time calculated in timeAdvance (Figure 5.42) elapses, transitioning to HouseHealthGeneratorStateProducing. During this transition, the person in the selected house is scheduled to experience a coronary event.

In HouseHealthGeneratorStateProducing, the filltimeAdvanceCoronary method (figure 5.43) is invoked during deltaInternal. This method generates a random time interval between zero and five years, representing the time until a person who has already experienced one coronary event might experience another. Subsequently, the houseID of the selected house is transmitted to the HouseHealthProcessor to simulate the event. This completes the intricate processes within the HouseHealthGenerator, which accurately emulates the occurrence of coronary events based on statistical parameters.

HouseHealthProcessor The HouseHealthProcessor plays a vital role in the simulation by evaluating health-related events and initiating alarms when required. This section provides insights into the inner workings of this compo-

```
void HouseHealthGenerator::calculateCoronary(){
    for(int i=0;i<numHouses;i++){
        double randPercent = distributionPercentage(*rng); int aMaxCoronary;
        if((*genderOfPeople)[i].compare("m")==0){//male
            if(randPercent<0.03) aMaxCoronary=50;
            else if(randPercent<0.069) aMaxCoronary=60;
            [..]
        }
        else{ //female
            if(randPercent<0.016) aMaxCoronary=50;
            [..]
        }
        int age=(*ageOfPeople)[i];
        int lowBorder=aMaxCoronary-age-10; int highBorder=aMaxCoronary-age;
        if(age>aMaxCoronary){
            highBorder=5; lowBorder=0;
        }
        else if(age>aMaxCoronary-10){
            highBorder=10; lowBorder=0;
        }
        distribution=Durationuniform_int_distribution<int>
          (lowBorder*secondsperyear, highBorder*secondsperyear);
        timeAdvanceCoronary[i]=distributionDuration(*rng);
    }
}
```

Figure 5.41.: *CalculateCoronary function of HouseHealthGenerator*

```
simtime_t HouseHealthGeneratorStateIdle::timeAdvance([...]ctx)
{
    auto gen = static_cast<HouseHealthGenerator*>(ctx);
    map<int,int> *timeAdvance = gen->getTimeAdvanceCoronary();
    int minTime=INT_MAX;int minHouse=0; // calculate time for next coronary
    for(int i=0;i<gen->numHouses;i++){
        if((*timeAdvance)[i]<minTime){
            minHouse=i; minTime=(*timeAdvance)[i];}
    }
    gen->actualHouse=minHouse;
    return minTime-simTime().dbl();
}
```

Figure 5.42.: *timeAdvance function of HouseHealthGeneratorStateIdle*

nent. The core of the HouseHealthProcessor is represented by HouseHealthProcessorStateAlarm, where it responds to health events. When an event of this nature occurs, it generates an output signal, specifically a SimEvent denoting a health event, to communicate with the rest of the simulation. The event is characterized by properties such as coronaryDetected and strokeDetected, which specify the type of health event. In the case of the HouseHealthProcessor, these are set to false, indicating an alarm for health events unrelated to coronary or stroke issues. The associated houseID is extracted and added to the SimEvent, providing crucial information about the location of the health event. This event is then included in the output bag, which enables communication between various components of the simulation. With this mechanism, the HouseHealthProcessor plays a pivotal role in responding to health-related alarms within the simulation, ensuring timely and appropriate actions are taken. This concludes the detailed description of the House Health model's implementation. The combination of the HouseHealthGenerator and HouseHealthProcessor accurately emulates health events in the simulated households, enriching the realism and depth of the simulation.

```
void HouseHealthGeneratorStateProducing::deltaInternal([..] ctx, [..] internalEvent)
{
    auto gen = static_cast<HouseHealthGenerator*>(ctx);
    gen->filltimeAdvanceCoronary();
    changeState(ctx, HouseHealthGeneratorStateIdle::getInstance());
}
void HouseHealthGenerator::filltimeAdvanceCoronary(){
    distributionDuration = std::uniform_int_distribution<int>(0, 5*secondsperyear);
    timeAdvanceCoronary[actualHouse]+=distributionDuration(*rng);
}
```

Figure 5.43.: *Call of filltimeAdvanceCoronary in HouseHealthGeneratorStateProducing*

5.4.4. Electronics

The Electronics module within the simulation encompasses various electronic devices found in households, each with its own unique attributes and behaviors. This section delves into the implementation details of this module, covering key aspects such as device management, state transitions, and event handling.

In the Electronics constructor (Figure 5.44), the basic parameters such as dayDuration, secondsPerYear, and numHouses are initialized. The rng variable is set up for random number generation. Subsequently, devices for each house are initialized, each with distinct attributes, such as power consumption and operating hours. These attributes are arbitrarily defined since there was no available empirical data for device behavior. The daysWorkPerWeek array is also initialized to specify the number of workdays for each house. The calculateDayTimes method (Figure 5.45) is responsible for determining the

```
Electronics::Electronics(int secondsPerYear,int numHouses)
{
this−>dayDuration=int(secondsPerYear/365);
    this−>dayDuration=int(secondsPerYear/365);
    this−>secondsPerYear=secondsPerYear;
    this−>numHouses=numHouses; this−>ageOfPeople=ageOfPeople;
    auto seed = std::chrono::system_clock::now().time_since_epoch().count();
    rng = new std::mt19937(seed);
    this−>currentState = ElectronicsStateIdle::getInstance();
    for(int i=0;i<numHouses;i++){
        devices[i].push_back(new Device("DishWasher", 0.4, 0.5, 1.5,true));
        devices[i].push_back(new Device("WashingMashine", 0.3, 1.5, 2.5,true));
        devices[i].push_back(new Device("TV", 0.6, 1, 2,false));
        devices[i].push_back(new Device("Mower", 0.05, 1, 2,true));
        devices[i].push_back(new Device("Computer", 0.6, 1, 3,false));
        devices[i].push_back(new Device("Cooker", 0.85, 0.10, 0.35,false));
        devices[i].push_back(new Device("Hover", 0.2, 0.5, 1.0,true));
        devices[i].push_back(new Device("Radio", 0.95, 2, 4,false));
        devices[i].push_back(new Device("LigthMorning", 10.0, 0.0, 1.5,false,true));
        devices[i].push_back(new Device("LigthEvening", 10.0, 2.0, 6.0,false,false,true));
        devices[i].push_back(new Device("Fridge", 10.0, 24, 24,true)); //always on
        devices[i].push_back(new Device("Freezer", 10.0, 24, 24,true)); //always on
        devices[i].push_back(new Device("Router", 10.0, 24, 24,true)); //always on
        daysWorkPerWeek[i]=5;
    }
    actualDevice=devices[0][0];
    fillElectronicNext();
}
```

Figure 5.44.: *Constructor of Electronics*

sleep and work times of the occupants. The function calculates when a person wakes up, leaves for work (if it's a working day), returns home, and goes to sleep.

```
void Electronics::calculateDayTimes(int actualHouse){
    int minStandUp=(6*dayDuration/24), maxStandUp =(8*dayDuration/24);
    int j=actualHouse;
    standUpTime[j] = getUniformDistributionNumber(minStandUp, maxStandUp);
    if(isWorkDay[j]){
      int minleavehouse= (7*dayDuration/24),maxleavehouse= (15*dayDuration/24);
      leaveHouseTime[j]=getUniformDistributionNumber(minleavehouse,maxleavehouse);
      int minduration= (6*dayDuration/24), maxduration= (10*dayDuration/24);
      int workduration = getUniformDistributionNumber(minduration,maxduration);
      comebackHomeTime[j] = leaveHouseTime[j]+workduration;
    }else{
      leaveHouseTime[j]=standUpTime[j]; comebackHomeTime[j]=standUpTime[j];
    }
    int minSleepTime= (22*dayDuration/24); int maxSleepTime= (24*dayDuration/24);
    sleepTime[j] = getUniformDistributionNumber(minSleepTime,maxSleepTime);
}
```

Figure 5.45.: *CalculateDayTimes method*

The ElectronicsChangeState function manages the state transitions of electronic devices. It evaluates whether a device should be turned on or off based on specific conditions. If a device is not active, it is turned on, and the time until it turns off is set. However, a special case is considered: when a device is already on, but the occupant leaves for work, indicating that the device should be turned on again when the occupant returns home. If the device is turned off for any other reason, it is marked as inactive.

```
double Electronics::calculateNextTimeAdvance(){
    actualDevicePair={"",INT_MAX};
    for(int i=0;i<numHouses;i++){
        auto actualmin= getMinTimeDevice(electronicsNextActive[i]);
        if(actualmin.second<actualDevicePair.second){
            actualDevicePair=actualmin;
            actualHouse=i;
        }
    }
    actualDevice= getDeviceByName(actualDevicePair.first, actualHouse);
    double simtime= simTime().dbl();
    double startDayTimeDouble= startDayTime.dbl();
    return actualDevicePair.second+startDayTimeDouble-simtime;
}
```

Figure 5.46.: *calculateNextTimeAdvance method of Electronics*

The calculateNextTimeAdvance function (Figure 5.46) is essential for managing the timing of electronic device activations. It calculates the time remaining until the next device event. It does so by identifying the next device to activate in all households, considering the start time of the day and the time until the device becomes active.

```
void ElectronicsStateProducing::outputFunction( ctx, internalEvent, outputBag){
    auto electronics = static_cast<Electronics*>(ctx);
    if(internalEvent.time−electronics−>startDayTime>=electronics−>dayDuration){
        SimElectronicChange simElectronicChange; //End of Day reached
        for(int j=0;j<electronics−>numHouses;j++){
            for(std::size_t i=0; i<electronics−>devices.size() and i<50; ++i){
            //current maximum of 50 electronic Devices
                simElectronicChange.electronicsActive[i]=false;
            }
            simElectronicChange.houseID=j; simElectronicChange.length=50;
            SimEvent event(simElectronicChange);
            PortValue<SimEvent,int> output(electronics−>alternateId,event);
            outputBag.insert(output);
        }
        electronics−>fillElectronicNext();
    }else{ //Not End of Day reached
        SimElectronicChange simElectronicChange;
        electronics−>ElectronicsChangeState();
        for(std::size_t i=0; i<electronics−>devices.size() and i<50; ++i){
            simElectronicChange.electronicsActive[i]
                =electronics−>electronicsActive[electronics−>actualHouse][i];
            simElectronicChange.houseID=electronics−>actualHouse;
            simElectronicChange.length=50;
        }
        SimEvent event(simElectronicChange);
        PortValue<SimEvent,int> output(electronics−>alternateId,event);
        outputBag.insert(output);
    }
}
```

Figure 5.47.: *outputFunction of ElectronicsStateProducing*

The outputFunction in ElectronicsStateProducing (Figure 5.47) manages the output of the Electronics module. It distinguishes between two cases: the end of the day and state changes in the Electronics module. When the end of the day is reached, it resets all devices in all households to an inactive state and prepares the devices for the next day. In the case of a state change, it calls the ElectronicsChangeState function (Figure 5.48) to update the states of electronic devices, then generates an output event that reflects these changes. This output event is crucial for communication between different components of the simulation.

These elements collectively form the foundation of the Electronics module, enabling the simulation to emulate the usage patterns of electronic devices within households, taking into account individual preferences, work schedules, and daily routines.

```
void Electronics::ElectronicsChangeState(){
    string name=actualDevice->name;
    int id=actualDevice->id;
    bool isActive = electronicsActive[actualHouse][id];
    if(!isActive){
        electronicsActive[actualHouse][id]=true;
        if(!actualDevice->secondOn){
          electronicsNextActive[actualHouse][name]=actualDevice->TimeTillOff;
        }else{ //Second time on because of work
          electronicsNextActive[actualHouse][name]=actualDevice->TimeTillOffSecond;
        }
    }else if(actualDevice->TimeTillOnSecond>0 and !actualDevice->secondOn){
    // Work time is during run time but actual before work
        actualDevice->secondOn=true;
        electronicsNextActive[actualHouse][name] = actualDevice->TimeTillOnSecond;
        electronicsActive[actualHouse][id]=false;
    }else{ // Device is turned off and not used this day anymore
        electronicsActive[actualHouse][id]=false;
        electronicsNextActive[actualHouse][name]=dayDuration+1;
        actualDevice->secondOn=false;
    }
}
```

Figure 5.48.: *ElectronicsChangeState method*

5.4.5. House Energy Calculation

The House Energy Calculation module within the simulation is responsible for modeling energy consumption in households. This section will provide an overview of the key components and functions involved in this module.

The HouseEnergyConsumer class (figure 5.49) is a fundamental part of the energy calculation module. It is designed to handle two states of energy consumption for various household devices. The initializePower function sets up the power consumption values for when the device is on and when it's in sleep or off mode. The updatePowerConsumption function allows for the dynamic updating of power consumption based on the current state of the device.

In the NodeGenerator class, the initializeEnergyBase function (figure 5.50) is responsible for initializing the energy calculation framework. It begins by defining the power consumption characteristics of various household devices and stores them in the devices vector. For each device, a corresponding HouseEnergyConsumer is created, and its power consumption attributes are set using the initializePower function. These consumers are then added to the energyBase, which serves as a repository for all energy consumers.

The calculateEnergyConsumption function (figure 5.51) is responsible for computing energy consumption within a household. It is called when the status of a device changes in Layer 2. This function updates the states of devices in the HouseEnergyConsumer objects and calculates the current energy consumption using the getTotalPowerConsumption method provided by the energyBase. It also computes various energy-related values and creates

```
void HouseEnergyConsumer::initializePower(W powerConsumptionOn,
  W powerConsumptionSleepOrOff, string name)
{
    this->powerConsumptionOn=powerConsumptionOn; this->isOn=false;
    this->powerConsumptionSleepOrOff=powerConsumptionSleepOrOff;
    this->name=name;
    powerConsumption = isOn ? powerConsumptionOn : powerConsumptionSleepOrOff;
}
void HouseEnergyConsumer::updatePowerConsumption()
{
    powerConsumption = isOn ? powerConsumptionOn : powerConsumptionSleepOrOff;
    emit(IEpEnergySource::powerConsumptionChangedSignal, powerConsumption.get());
}
```

Figure 5.49.: *Part of the HouseEnergyConsumer*

```
void NodeGenerator::initializeEnergyBase(){
    std::vector<TDevice> devices;
    [..] // add PowerConsumption of the various devices
    devices.push_back({"Computer",W(250),W(10)}); // 1. name, 2. on, 3. standby
    devices.push_back({"Cooker",W(4500),W(0)});
    [..]
    energyBase=new EpEnergySourceBase(); numDevices=devices.size();
    for(std::size_t i=0;i<(size_t)numDevices;i++){
        HouseEnergyConsumer *consumer = new HouseEnergyConsumer();
        consumer->initializePower(devices.at(i).powerConsumption,
        devices.at(i).standbyPowerConsumption, devices.at(i).name);
        energyBase->addEnergyConsumer(consumer);
        consumers.push_back(consumer);
    }
}
```

Figure 5.50.: *InitializeEnergyBase function of NodeGenerator*

a SimEvent to represent power consumption. Finally, this event is sent back to Layer 2 for further processing. In summary, the House Energy Calculation module models energy consumption within households, allowing devices to be in different states with varying power consumption. It provides a framework for tracking energy usage, updating device states, and communicating power consumption information to other parts of the simulation.

5.5. Implementation of the Human-in-the-loop Interface

The advent of sophisticated simulation frameworks has propelled the study and analysis of complex systems to new heights. However, to imbue these simulations with greater realism and applicability, the integration of human decision-making and interaction becomes paramount. This chapter delves into

```
void NodeGenerator::calculateEnergyConsumption(int currentHouseID){
    for(int i=0;i<numDevices;i++){
        bool active=electronicActive[i];
        HouseEnergyConsumer *consumerHouse=consumers.at(i);
        consumerHouse->setState(active);
    }
    tHouse = simTime(); tIntervalHouse=tHouse-tLastHouse[currentHouseID];
    tLastHouse[currentHouseID]=simTime();
    [..]
    inet::power::W currentEnergy = energyBase->getTotalPowerConsumption();
    double scale=secondsPerYear/(365*24*60);
    double currentPower=0.5*(currentEnergy.get()
      +lastEnergy[currentHouseID])*tIntervalHouse.dbl()*scale/60;
    lastEnergy[currentHouseID]=currentEnergy.get();
    powerSum[currentHouseID]+=currentPower; SimHousePower demand;
    demand.currentEnergyConsumption=currentEnergy.get(); // current energyConsumption
    demand.currentPower=currentPower; //Power since last massage
    demand.totalPower=powerSum[currentHouseID];// total power consumed
    demand.houseID=currentHouseID;
    SimEvent powerEvent = SimEvent(demand);
    SimEventMessage* smsg = new SimEventMessage("HousePowerdemand");
    smsg->setSimEvent(powerEvent);
    send(smsg, "amcOut");
}
```

Figure 5.51.: *CalculateEnergyConsumption function of NodeGenerator*

the meticulous design and realization of the Human-in-the-Loop (HITL) interface, an instrumental component of our simulation framework. The infusion of human elements into the simulation ecosystem introduces a host of intricate challenges and transformative prospects. It empowers researchers and practitioners to scrutinize, assess, and validate intricate system behaviors under the influence of diverse human-centric scenarios. This chapter scrutinizes the methodologies, technologies, and strategies pivotal in bridging the virtual and human domains seamlessly. Our exploration commences with a comprehensive elucidation of the foundational tenets governing the HITL interface. We elucidate its inherent purpose, multifaceted functionalities, and overarching objectives within the simulation framework. Subsequently, we embark upon a technical odyssey, dissecting the nuanced intricacies underpinning its practical implementation, including the crafting of user interfaces, establishment of data exchange protocols, and the intricate dynamics of user interaction. Moreover, we dissect the pivotal role of feedback mechanisms within HITL simulations, expounding upon their ability to facilitate a bidirectional exchange of information between human agents and the simulation milieu. Furthermore, we investigate the ethical and practical dimensions entailed in the incorporation of human participants into simulations. Throughout this chapter, we underscore the methodologies wielded for rigorous testing, validation, and optimization of the HITL interface. We accentuate the significance of usability studies and participant feedback loops in the iterative refinement of the interface, aligning it with the evolving needs and expectations of simulation end-users. In synthesis, the deployment of the Human-in-the-Loop interface represents more than a mere technological pursuit. It signifies a profound endeavor to harmonize the capabilities of advanced computational models with the intricacies of human decision-making within complex systems. The outcome is a simulation framework that transcends algorithmic modeling, allowing scholars to scrutinize the intricate interplay between technological systems and human behaviors within dynamic, multifaceted environments.

5.5.1. OMNeT++ Adapter Set-Up

Adapter Integration within OMNeT++

The OMNeT++ adapter presents itself as an adaptable solution, highly amenable to diverse simulation environments. This versatility simplifies the integration of the adapter into various simulations with minimal overhead. The OMNeT++ adapter serves as the cornerstone for external communication within OMNeT++ simulations.

Establishing Connection Points

One critical initial step in establishing an effective connection is determining the precise destination for data transmission. This fundamental configuration was addressed in section 4.8, where the specific IP address was identified. As the adapter operates as an integral component of the virtual operating system (OS), further detail is required. Drawing from the concepts elucidated in section 4.8 regarding TCP/IP communications, this detail is encapsulated within a port number.

Enabling HTTP Communication

With the full address confirmed, the OMNeT++ adapter becomes accessible for processing HTTP requests and responses. The adapter has been diligently designed to align with the structure of the VICINITY RESTful API, as documented in [102]. Moreover, as the OMNeT++ adapter functions as a RESTful API itself, it is proficient in processing JSON data, ensuring seamless interoperability. For the sake of consistency and modularity, the parking simulation will adopt the precise JSON data structure as mandated by the VICINITY API. Nonetheless, it's essential to highlight that the accepted HTTP requests within this context follow the pattern detailed in Listing 5.1.

Listing 5.1: *JSON objects example*

```
GET /adapter/objects
GET /adapter/objects/{oid}/properties/{pid}
PUT /adapter/objects/{oid}/properties/{pid}
```

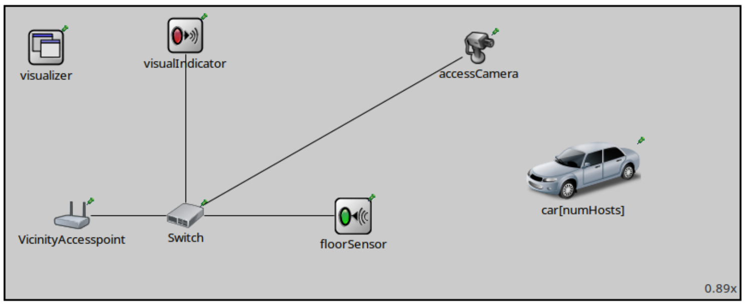


Figure 5.52.: *Graphical set-up of the simulation*

In the context of Listing 5.1, the URI lacks an IP address for clarity, although it is an inherent part of the overall structure. The request structure comprises two essential lines of inquiry:

1. **Object Discovery Data**: Primarily employed for debugging purposes, this request seeks to retrieve object names as a form of acknowledgment.
2. **Property Reading and Writing**: These requests are vital for the accomplishment of the research objectives. They facilitate both the retrieval and modification of property attributes. The unique Object Identifier (OID) and Property Identifier (PID) act as variables, rendering the URI adaptive to specific objects and properties.

The subsequent terms denote the path structure, akin to branches in a tree. Port number 4242 serves as the entry point, guiding the traversal from adapter to objects. Herein, objects become accessible, each identified by its unique Object Identifier (OID). Once a specific object is accessed, various possibilities unfold, albeit the path leading to property attributes is the primary concern. This path terminates with an array of properties, culminating in the selection of a particular property via a unique Property Identifier (PID). In summary, the systematic establishment of the OMNeT++ adapter integrates seamlessly within the overarching simulation architecture. It encompasses foundational aspects such as IP address, port number configurations, and URI structures, thereby providing the essential infrastructure for HTTP-based communications within HITL simulations.

5.5.2. Parking Simulation Set-Up

Building upon the groundwork established in Section 5.5.1 regarding the OMNeT++ adapter setup, this section delves into the intricacies of the parking simulation. To realize the concept outlined in Section 4.8 and visually represented in Figure 4.22, a detailed simulation setup is essential. Figure 5.52 provides a snapshot of this configuration.

The diagram in Figure 5.52 seamlessly marries the schematic illustration from Figure 4.22 with the theoretical framework elucidated in Section 4.8. Notably, it includes the representation of the access point of the OMNeT++

adapter. For the sake of achieving a higher degree of realism, a network switch is introduced, interposed between the adapter's access point and the IoT devices. This switch serves a straightforward purpose of facilitating the forwarding of requests and responses.

Assignment of Device Tasks

The parking simulation mandates the allocation of responsibilities to the visual indicator light and the floor sensor devices. To determine the occupancy status of a parking space, the floor sensor invariably serves as the queried device. Consequently, the Floor Sensor device always plays the role of the source in occupancy inquiries. In contrast, the publication of a reserved parking space is a joint operation involving both the visual indicator and the floor sensor devices. In this context, the visual indicator and floor sensor properties collectively represent the occupancy status. To simplify communication within the simulation, these properties are based on an off-the-shelf I/O device. This I/O device boasts the requisite functionalities of accessibility and modifiability in its state.

Object Identifiers (OIDs) and Property Identifiers (PIDs)

In this context, the names of the entities featured in Figure 5.52, such as visualIndicator, serve as unique Object Identifiers (OIDs). These OIDs correspond to specific entities within the simulation, representing distinct parking spaces, devices, or sensors. Furthermore, the term 'state' is adopted as a Property Identifier (PID) for each device, allowing it to express two discrete values: zero or one. A value of zero signifies that the respective parking space remains unoccupied, indicating that no vehicle currently occupies it. Conversely, a value of one communicates that the parking space has been reserved and is no longer available for parking.

Additional Simulation Elements

For the sake of completeness and to consider potential future developments, the simulation incorporates two additional components: a camera for parking space monitoring and a virtual vehicle. These elements, while currently devoid of specific functions, offer the flexibility to explore various scenarios and functionalities in subsequent stages of this research.

In essence, the parking simulation is constructed as a holistic human-in-the-loop framework that encompasses the visual representation of parking spaces, sensors, indicators, and communication infrastructure. This setup provides the foundational architecture required for conducting experiments and evaluations in the context of the Human-in-the-Loop (HITL) simulation framework.

5.5.3. *Digression*: iOS App Development

Architectural Design Pattern Decision

The development of the iOS application for the smart parking use case encompasses a spectrum of software features, including an intuitive user interface and a robust network communication channel. It's important to emphasize that this application was constructed from the ground up, necessitating careful considerations not only for functional aspects but also for non-functional trade-offs. A crucial initial step in this endeavor was the selection of an architectural design pattern. Native iOS applications are developed within Apple's integrated development environment, Xcode, and employ the Swift programming language. To seamlessly translate the wireframe design illustrated in Figure 5.59 into a fully functional application, an architectural blueprint was chosen.

Native iOS Development

Native iOS development is pivotal to the creation of an application that seamlessly integrates with the iOS ecosystem. Leveraging Apple's development tools and Swift language ensures a robust and optimized user experience for iOS users. This choice is essential for achieving a high level of compatibility, performance, and user-friendliness.

Translating Wireframes to Real App

The wireframe presented in Figure 4.24 serves as the foundational blueprint for the iOS application. The transition from concept to reality involves translating this wireframe into tangible app components, including user interface elements and interactive features. This process requires meticulous attention to detail and adherence to design guidelines to ensure a cohesive and intuitive user experience.

Establishing the Network Communication Channel

A critical component of the iOS application is its network communication channel, which enables seamless interaction with the simulation framework. This aspect is pivotal for real-time updates on parking space availability and reservation status. Technical solutions for establishing and maintaining this communication channel are introduced, ensuring robust and reliable data exchange between the application and the simulation environment (see figure 4.25).

In essence, the iOS application is an integral part of the Human-in-the-Loop (HITL) simulation framework, serving as the primary interface for users to access and interact with the smart parking system. The following sections delve into the architectural design, development process, and technical aspects of this iOS application.

Architectural Design Pattern

Architectural design patterns, by their nature, transcend specific programming languages and platforms, offering a best-practice approach for structuring software systems. In the realm of computer science, a multitude of architectural design patterns exists, each with its unique focus on enhancing code maintainability, facilitating unit testing, and improving code comprehensibility. However, for this evaluation, the focus is solely on architectural design patterns that pertain to the integration of a user interface with an application domain. In particular, this exploration is guided by the context of mobile applications, where distinctive patterns emerge due to the considerations of touchscreen interactions and the dynamic nature of mobile devices. Neglecting the use of such patterns can have substantial consequences. Complex codebases, without the guiding structure of architectural patterns, can result in significant overhead when striving to maintain consistency between the user interface and the application functionalities. Given the development context of this work, which involves the creation of an iOS application, two prominent architectural design patterns are noteworthy: the Model-View-Controller (MVC) pattern and the Model-View-View Model (MVVM) pattern (refer to Figure 5.53). Both of these patterns adhere to the fundamental principle of segregating the business-logic model from the user interface design. This clear separation of concerns not only enhances maintainability but also streamlines the division of labor among developers.

Model		View-Model/Controller		View
user		user		user
password	$\longleftrightarrow$	password	$\longleftrightarrow$	password
		confirm password		**confirm password**

Figure 5.53.: *Exemplification why it is important to use either MVC or MVVM pattern: The act of confirming the chosen password is very important because it preservers the user from typos. But the logic, which is responsible for storing the user data, is not interested in a confirmed password.*

Model-View-Controller (MVC) Pattern The Model-View-Controller (MVC) pattern, illustrated in Figure 5.54, is a widely employed architectural pattern for the development of iOS applications using the UIKit Framework within Xcode. This pattern aligns seamlessly with the iOS development workflow, wherein storyboards are employed to manage the control flow. Furthermore, Apple's developer documentation offers comprehensive insights into the synergy between MVC and UIKit [103]. The MVC pattern organizes the application into three distinct components, each with its specific role:

- **View**: The View component is responsible solely for rendering data on the smartphone's screen. It manages the presentation layer, which en-

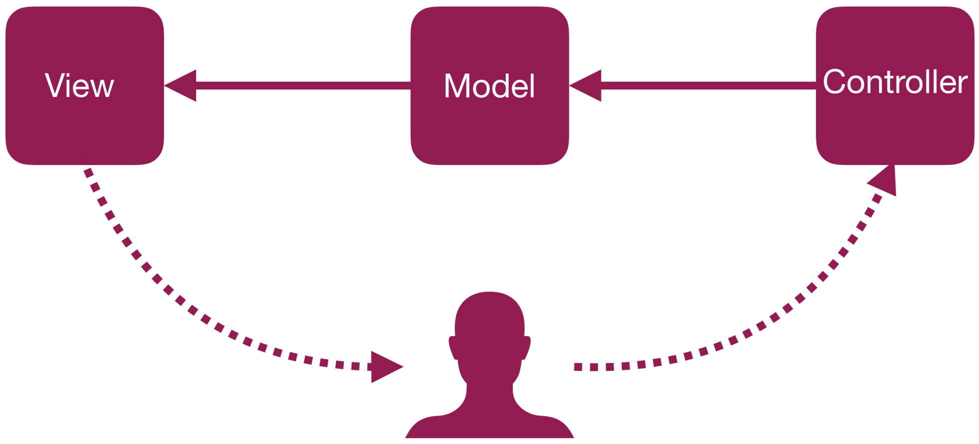


Figure 5.54.: *MVC architectural design pattern*

compasses elements such as text boxes and graphical elements intended for user interaction.

- **Controller**: Operating in tandem with the View component, the Controller is responsible for capturing user inputs and processing them. It acts as an intermediary between the user and the application's logic. In this capacity, the Controller not only manages the user's interactions but also undertakes more intricate tasks, such as networking operations.
- **Model**: The Model component encapsulates both the application's data and its business logic. It serves as the backbone of the application, maintaining data that is accessible to both the View and Controller layers.

However, while MVC offers a well-defined structure for iOS application development, it exhibits certain inherent limitations. One such limitation is highlighted in the conference paper "A Journey Through the Land of Model-View-* Design Patterns" [104], where a scenario involving the coloration of a text field in a financial report, based on varying values, underscores a challenge. MVC, by design, does not provide explicit mechanisms for handling specific states that are not inherently part of the View. To address such scenarios, developers may need to create custom views, which can introduce complexities when working with user interfaces.

Model-View-ViewModel (MVVM) Pattern The Model-View-View-Model (MVVM) architectural design pattern, depicted in Figure 5.55, presents a distinct approach compared to the previously mentioned MVC pattern. Notably, in June 2019, Apple introduced SwiftUI, a new framework within Xcode, during its Worldwide Developer Conference (WWDC). While SwiftUI is not positioned as a direct successor to UIKit, it offers a compelling alternative for iOS app development. The distinguishing feature of SwiftUI is its declarative programming paradigm, which contrasts with the imperative programming

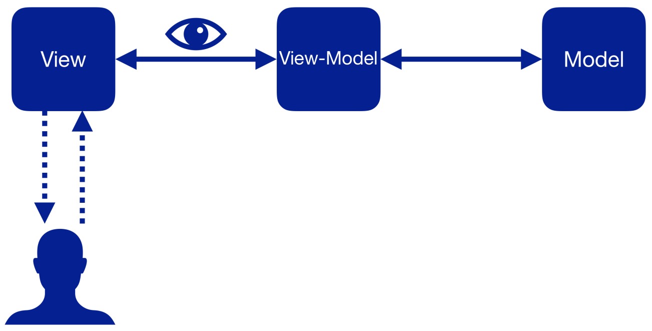


Figure 5.55.: *MVVM architectural design pattern*

employed in UIKit. In declarative programming, views are constructed as functions of their state, eliminating the need for explicit code to manage the view's behavior. Data binding at runtime is intrinsic to SwiftUI, reducing reliance on manual storyboard editing. In SwiftUI, views are driven by their state, which dictates their presentation. This shift from imperative to declarative programming has transformative implications for UI development. In the MVVM pattern, the three core components operate as follows:

- **View**: In MVVM, the View component assumes a more active role by both displaying data and processing user input. Additionally, it adopts an observing synchronization mechanism. This synchronization is established through subscriptions to the ViewModel. The declarative data binding inherent to SwiftUI enables this observing synchronization, facilitating separation between the View and ViewModel layers.

- **View-Model**: The ViewModel, in conjunction with the View, orchestrates the communication between the View and Model layers. It notifies the View of any changes through a notification mechanism, thus enabling the observing synchronization mentioned earlier. The ViewModel acts as a mediator and manages tasks such as networking operations. Importantly, it can preprocess model input before delivering it to the View. This capability allows the ViewModel to provide the View with different view states, which is a notable departure from the MVC pattern. The concept of view states equips the ViewModel with additional logic.

- **Model**: The Model component maintains and manages data, along with the core business logic of the application. It remains agnostic of the View and ViewModel layers.

The MVVM pattern, as exemplified [104], offers advantages such as the ViewModel's ability to call upon the Model for data retrieval and processing in different view states. Each View possesses its own logic for presenting data, empowering the ViewModel with diverse ways to interact with the Model. In this iOS App development process, the Model-View-View-Model (MVVM) architectural design pattern was selected due to the promising future of the

SwiftUI framework in the broader Apple ecosystem. SwiftUI unifies app development across various Apple devices, influencing the future of VICINITY's work. Additionally, the adoption of declarative programming with data binding exemplifies modern software development practices.

Pattern Selection and User Experience It is crucial to note that the selection of an architectural design pattern does not impact the final user experience of the application. These patterns primarily influence the development process, ensuring code quality, maintainability, and scalability. Regardless of the chosen pattern, the end user interacts with the application without being aware of the underlying architectural structure.

In summary, the decision to adopt either the MVC or MVVM architectural design pattern is pivotal for optimizing the development process of the iOS application. These patterns offer clear guidelines for organizing code, facilitating collaboration among developers, and ensuring a robust and maintainable software solution. The following subsections are structured to align with the MVVM pattern's principles.

View: User Interface The development of a real iOS graphical user interface builds upon the wireframe presented in section 4.8. This wireframe serves as the foundation for creating a user-friendly and visually appealing iOS app. To ensure an effective user experience (UX), adherence to Apple's Human Interface Guidelines (HIG) is paramount [105]. These guidelines provide comprehensive insights and resources for designing apps that seamlessly integrate with Apple's platforms. In accordance with the components identified in Figure 4.24, let's evaluate how these design principles are applied to the various app elements: text fields, images, actions, and navigation.

Text Fields and Images Text fields and images within the app adhere to fundamental principles of legibility and consistency. Text is presented in a legible black font, aligned left for optimal readability. Images are consistently high-resolution, ensuring they appear sharp and clear on the device's screen. These recommendations align seamlessly with the wireframe presented in Figure 4.24.

Performing Actions Action elements in the app require careful consideration, as they can significantly impact the user's experience. Apple's HIG advises developers to use action features judiciously, preventing distractions and disconnection from the main app flow. Animation effects, if employed, should align with real-world physics, ensuring they are natural and intuitive. In the context of this work, the animation of a car's movement serves as an apt example. This movement is visualized through a slider element, which effectively communicates the concept of parking space occupancy (state transitioning from 0 to 1). The consideration of animation principles in this context creates a more engaging and realistic user experience. This concept also aligns with the 'state' property described in Section 5.5.2, emphasizing the app's commitment to conveying information effectively.

Navigation App navigation is an essential architectural component that influences how users interact with the app. Apple's HIG presents three primary navigation approaches: hierarchical navigation, flat navigation, and content-driven navigation.

- **Hierarchical Navigation** (Figure 5.56): This approach resembles a tree structure, where users start with one choice per screen and navigate step by step. To make different choices, users must retrace their steps. This hierarchical navigation style may not align with the app's requirements, as outlined in the functional specification.
- **Flat Navigation** (Figure 5.57): The flat navigation approach allows for several initial choices that users can switch between effortlessly. Each choice leads to a specific view, facilitating easy exploration of multiple content categories. The flat navigation style aligns well with the app's functional requirements, as evidenced in Figure 4.24.
- **Content-Driven or Experience-Driven Navigation** (Figure 5.58): This approach encourages users to explore the app at their own pace, facilitating content discovery. While suitable for certain app types like gaming or books, it may not be the most appropriate choice for this project.

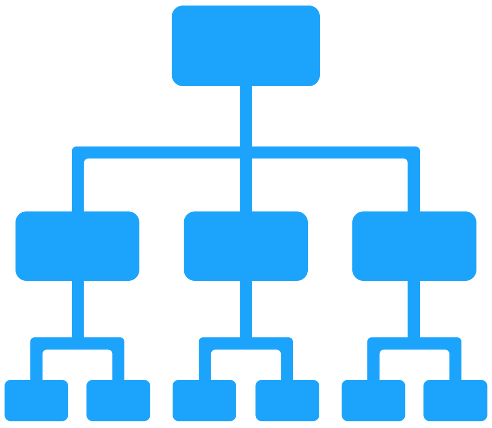


Figure 5.56.: *Hierarchical navigation approach.*

Considering the app's functional specification and the wireframe composition, the flat navigation approach (Figure 5.57) aligns most closely with the app's requirements. Notably, this decision is reinforced by the app's wireframe, which inherently follows a flat navigation structure. Furthermore, Apple's Xcode framework provides tab bars, which offer a practical navigation solution. Tab bars, typically located at the bottom of an app screen, enable

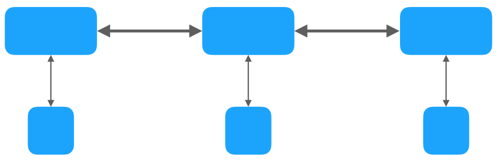


Figure 5.57.: *Flat navigation approach*

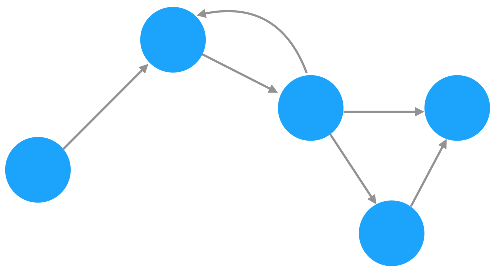


Figure 5.58.: *Content or experience driven navigation*

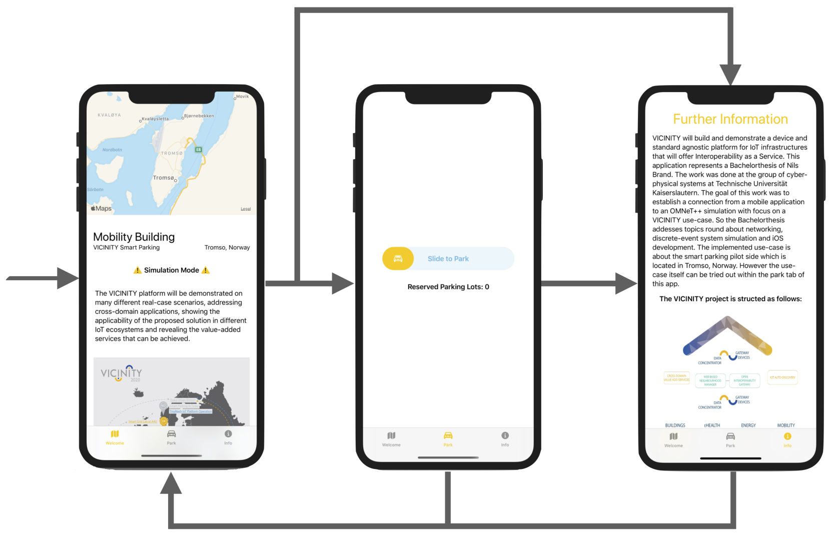
users to switch swiftly between different app categories, enhancing usability and accessibility.

The user interface of the app is visualized in Figure 5.59, extending the wireframe concept into a fully realized interface. The welcome screen closely resembles its wireframe counterpart, with minimal changes for user interaction clarity. An interactive map replaces the top image, providing users with a responsive UX element. The middle tab guides users to the action-performance view, maintaining a minimal interface design to emphasize interaction with the slider element. The last tab, consistent with the wireframe, provides users with detailed app information, including an imprint. To maintain brand consistency with VICINITY, the app incorporates VICINITY's colors and adopts an app icon that encapsulates the brand's design language. This cohesive approach ensures a seamless transition from wireframe to a fully functional user interface, aligning with Apple's HIG recommendations.

Model-View: Networking In this work, the core responsibilities of the Model-View component encompass networking tasks and the encoding/decoding of JSON data, akin to the functionalities performed by a web browser when interacting with a RESTful API. Upon launching the app, one of the initial requests executed is a GET command, which furnishes the user with information about parking space availability. Building upon the setup described in Sections 5.5.1 and 5.5.2, the absolute path for this web service is as follows:

```
http://192.168.56.104:4242/adapter/objects/floorSensor/properties/state
```

Here, the OID (Object Identifier) corresponds to floorSensor as detailed in Section 5.5.1. Similarly, the requested PID (Property Identifier) is the state property. The web service responsible for fulfilling the GET request also handles potential URIs (Uniform Resource Identifiers) session interceptions, a best practice paradigm often used in internet communication. Upon receiving data, the final step in the networking component's GET request is to decode the



The VICINITY platform will be demonstrated on many different real-case scenarios, addressing cross-domain applications, showing the applicability of the proposed solution in different IoT ecosystems and revealing the value-added services that can be achieved.

VICINITY will build and demonstrate a device and standard agnostic platform for IoT infrastructures that will offer Interoperability as a Service. This application represents a Bachelorthesis of Nils Brand. The work was done at the group of cyber-physical systems at Technische Universität Kaiserslautern. The goal of this work was to establish a connection from a mobile application to an OMNeT++ simulation with focus on a VICINITY use-case. So the Bachelorthesis addesses topics round about networking, discrete-event system simulation and iOS development. The implemented use-case is about the smart parking pilot side which is located in Tromso, Norway. However the use-case itself can be tried out within the park tab of this app.

Figure 5.59.: *Mockup of the app which also represents the realization of the wireframe*

JSON data, making it accessible for further processing within the Swift codebase and subsequently rendering it in the app's view. When a user initiates the parking process, data from the smartphone must traverse the network to reach the simulation. This involves data transmission through various nodes, as depicted in Figure 4.25. As established in Section 5.5.1, to ensure that other participants can observe the parking space reservation, two PUT requests are dispatched: one for the visual indicator and one for the floor sensor. The PUT request addresses are as follows:

```
  http://192.168.56.104:4242/adapter/objects/floorSensor/properties/state
http://192.168.56.104:4242/adapter/objects/visualIndicator/properties/state
```

Once again, the chosen communication protocol is HTTP. In terms of OID and PID, the principles mirror those outlined in the previous description, with the distinction that this pertains to the PUT command and the addressing of two distinct objects. In this context, the responsibility of Swift is to serialize the data into JSON format, making it suitable for processing by the OMNeT++ adapter. It's important to highlight that throughout the execution of the MVVM architecture, the Model-View component also manages all bindings, states, and subscribed notifications, ensuring the smooth flow of data during runtime.

Model: Business-Logic Although the Model manages a relatively limited variety of business logics, the complexity arises from the in-depth understanding of the utilized business logic. This complexity is a direct result of adhering to the guidelines for designing Thing Descriptions for VICINITY integrators [106], which are based on the Web of Things (WoT) Thing Description working draft of W3C [107]. The WoT Thing Description format aims to provide a formal model for a common representation of WoT, detailing how metadata and interfaces of things can be employed for interoperability among devices or applications using the JSON format. However, not every parameter from VICINITY's Thing Descriptions is needed in the app's model, leading to a simplified and more comprehensible model for iOS app development. The following tables elucidate the significant values for the model. Starting from a top-down perspective, the outermost enclosure provides information about the adapter type, referred to as the adapter-id' in Table 5.2. In the context of this app, it signifies the OMNeT++ adapter. Furthermore, Table 5.2 represents the complete serialization of Thing Descriptions due to the identically named array of objects. As outlined in the command GET /adapter/objects', it directly leads to the comprehensive Thing Descriptions in JSON format.

Field name	JSON Construct	Description
adapter-id	string	Unique identifier of adapter within theagent service.
thing-descriptions	array of objects	The array of thing descriptions.

Table 5.2.: *Serialization of Thing-Descriptions in VICINITY*

Advancing a step further and delving into the Thing Descriptions array concerning objects, Table 5.3 showcases the fields of interest. The 'oid' (Object Identifier) is the equivalent of the command 'GET /adapter/objects/{oid}'. The 'oid' and 'name' fields are not employed in the app's visualization but may be contemplated for future developments.

Field name	JSON Construct	Description
oid	string	Infrastructure specific unique identifier of the object
name	string	Human readable name of object, visible in neighbourhood manager
properties	array of objects	The array of property interaction patterns see Property

Table 5.3.: *Serialization of objects in VICINITY*

Moving on to the properties array, Table 5.4 provides access to the 'pid' (Property Identifier), which is the exact property inspected to ascertain parking space availability and to simulate vehicle parking within the system. Therefore, the path to this JSON data involves the 'GET' and 'PUT' commands:
`/adapter/objects/{oid}/properties/{pid}`.

Field name	JSON Construct	Description
pid	string	Unique identifier of the property.

Table 5.4.: *Serialization of a property in VICINITY*

For the iOS app, this translates to the following unwrapping steps after establishing a connection with the adapter:

1. Access Thing Descriptions: Displayed objects include visualIndicator, floorSensor, and accessCamera.

2. Upon selecting one of the oids, the properties array becomes accessible.

3. Within this array, the unique pid, referred to as state in Section 5.5.2, contains the desired information.

5.6. Evaluation

The Evaluation chapter presents a rigorous assessment of the Smart Home Simulation System. This evaluation is structured into three pivotal components: Performance Evaluation, Software-in-the-Loop (SIL) Testing, and Human-in-the-Loop (HIL) Testing. Each component adheres to precise methodologies and metrics, aimed at scrutinizing diverse facets of the system's performance, functionality, and user experience. The primary focus of the inaugural section resides in the meticulous examination of system performance. In the realm of complex systems, such as smart homes, performance is a linchpin attribute. This section employs a multifaceted approach encompassing metrics related to computational efficiency, scalability, and responsiveness. By rigorously evaluating these parameters, we seek to quantify the system's capability to manage increasing loads, maintain real-time responsiveness, and optimize computational resources. The second section gravitates towards Software-in-the-Loop (SIL) Testing, an indispensable phase for isolating and scrutinizing the software components in isolation. This phase plays a pivotal role in identifying and rectifying software-related anomalies. Through structured testing scenarios, we verify the correctness and efficiency of each software module, ensuring that they seamlessly integrate and function harmoniously within the overarching system. The third section is dedicated to the intricate domain of Human-in-the-Loop (HIL) Testing. This testing regimen injects real or simulated users into the system to gauge its human-facing attributes. User-centric evaluations encompass usability, user-friendliness, and overall user experience. Insights garnered from HIL testing provide critical feedback, enabling enhancements in user interface design and functionality to augment user satisfaction and system usability. These three evaluative components are designed to yield a comprehensive understanding of the Smart Home Simulation System's capacities and constraints. The outcomes of these assessments will serve as pivotal directives for refining and fortifying the system, driving it towards the pinnacle of performance, reliability, and user-centric design.

5.6.1. SysMD Model Evaluation

SysMD presents a formal and textual modeling language that empowers designers to effectively capture system characteristics such as inheritance, decomposition, and their associated dependencies. SysMD's versatility is evident in its applicability to a wide range of domains, including mechatronics, physics, hardware, software, and Analog/Mixed-Signal systems. Moreover, SysMD seamlessly integrates documentation within the model, enabling comprehensive knowledge bases. These knowledge bases serve as the foundation for defining intricate system designs, which can be readily evaluated for early performance estimates. The language's accessibility, particularly to domain experts, is a significant advantage. SysMD's user-friendly nature, achieved through the use of semantic triples and compatibility with SysMLv2 textual representations, simplifies the modeling process. Additionally, the incorporation of units and the concept of quantities introduces an additional layer of model checking, enhancing consistency and reliability. The SmartGrid use case presented in this dissertation exemplifies how SysMD can be applied to assess various system properties across different scenarios. Notably, SysMD's models are highly readable and interpretable, even for individuals lacking expertise in modeling. This stands as a significant advantage over SysMLv2 with OCL, making SysMD a valuable tool for system modeling and analysis.

5.6.2. Performance Evaluation

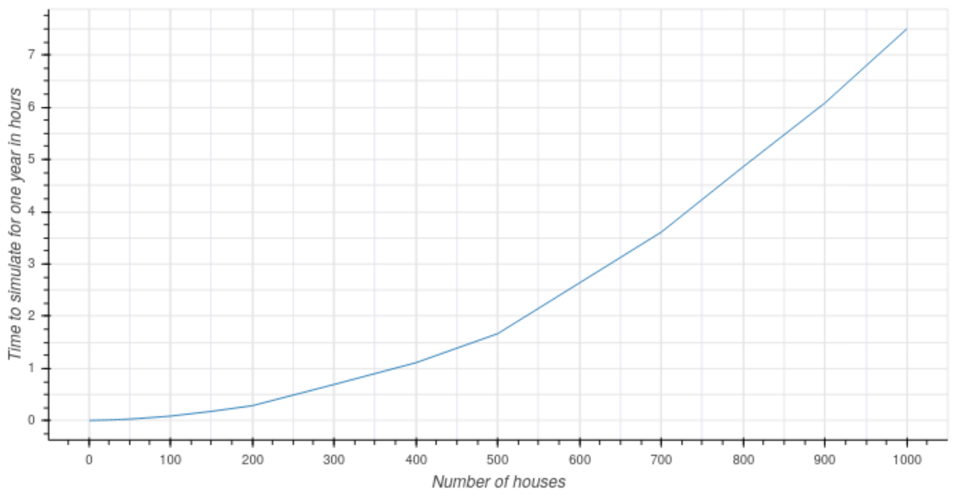


Figure 5.60.: *Performance for the simulation in one year*

Performance evaluation represents a critical dimension in the assessment of the Smart Home Simulation System's efficiency and scalability. This chapter meticulously dissects the system's performance characteristics, focusing on the relationship between computational resources, simulation time, and the number of houses in the model.

Experimental Setup The cornerstone of this evaluation lies in the precise configuration of simulation parameters. The variable denoted as *secondsPerYear* within the *HouseNetwork* module was meticulously set to 3652460, effectively establishing a real-time simulation framework. In this context, each simulation second equates to one real-world minute. This temporal calibration not only enables meaningful performance measurements but also ensures the system's suitability for real-time execution.

Real-Time Simulation Validation To ascertain the system's capability to maintain real-time responsiveness, a series of experiments were conducted. Foremost among these experiments was the measurement of simulation time required for the completion of one virtual year, with variations in the number of houses in the simulation model. The outcomes of these experiments are graphically presented in Figure 5.60.

Performance Observations Figure 5.60 portrays a compelling narrative regarding the system's performance. Notably, the simulation's temporal efficiency surpasses real-time constraints. For instance, simulating a single house over the course of a year consumes a mere 22 seconds. However, as the number of houses in the simulation increases, computational demands scale proportionally. Intriguingly, the computational effort grows at a rate that is notably less than linear. For instance, simulating 1000 houses extends the duration to a modest 7.5 hours, which is merely a thousand-fold increase compared to a single house. This observation underscores the system's capacity to efficiently simulate a substantial number of houses, substantiating its scalability. The adherence to real-time simulation benchmarks is of paramount significance, as it ensures that the system can be executed in real-time scenarios, such as for monitoring, decision support, or training purposes. The observations presented herein lay the foundation for a thorough understanding of the system's performance attributes, vital for its real-world applicability.

House Energy Consumption

This section scrutinizes the electricity consumption patterns exhibited by the simulated houses. To investigate the dynamic trajectory of energy utilization, three distinct experiments were conducted, each designed to represent specific scenarios. The results of these experiments, presented in Figure 5.61, Figure 5.62, and Figure 5.63, delineate the daily energy consumption profiles of 1000 houses over a simulated day.

Energy Consumption on Average Days Figure 5.61 illustrates the energy consumption profile on a typical day, incorporating the randomness inherent in real-world human activities. In this simulation, residents in these houses follow a weekly schedule that consists of randomly distributed workdays and days off, with a frequency of five workdays and two non-working days per week.

During the early hours of the day, from midnight to 6 a.m., the simulation mirrors the sleeping patterns of the residents, resulting in a relatively constant energy consumption level. This consistency arises from essential devices such as refrigerators, freezers, and routers, which remain operational throughout

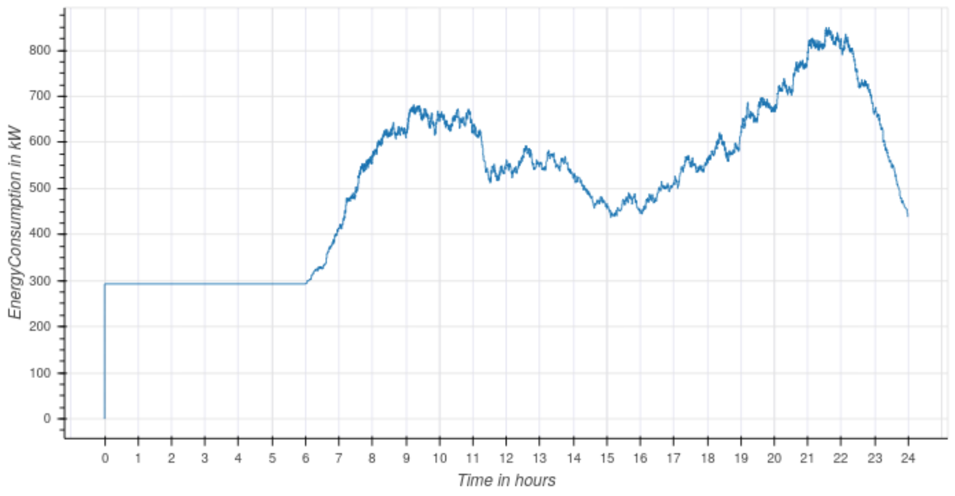


Figure 5.61.: *Energy Consumption of 1000 houses on average days*

the night. The gradual rise in energy consumption commences as residents awaken, engage in activities, and initiate device usage. Notably, the energy demand surges as residents prepare for work or other daily responsibilities, driving a noticeable peak. Subsequently, as some residents depart for work, there is a corresponding decline in energy consumption. However, due to the presence of residents who remain at home, the energy consumption remains elevated compared to the nighttime baseline. In the evening, upon the return of residents, a further spike in energy consumption materializes, primarily attributable to increased electronic device usage and heightened lighting requirements. The cycle concludes with a reduction in energy usage after 10 p.m., coinciding with residents retiring to bed.

Energy Consumption on Working Days In Figure 5.62, the simulation is configured to emulate only working days for the 1000 residents. This scenario reveals distinctive energy consumption characteristics that reflect the demands of a typical workday.

Two prominent peaks in energy consumption are evident, occurring during the morning and evening hours. The first, occurring around the morning hours, stems from residents awakening, engaging in early tasks, and notably, illuminating their surroundings. However, it is the evening peak that stands out, driven primarily by the increased demand for lighting during the evening hours. Importantly, at approximately 4 p.m., energy consumption plummets to levels akin to those observed during nighttime. This abrupt dip is associated with the majority of residents being at their workplaces.

Energy Consumption on Non-working Days The final experiment, depicted in Figure 5.63, recreates a scenario in which none of the 1000 residents engage in work-related activities. This configuration emphasizes increased leisure time and its consequent effect on energy consumption patterns.

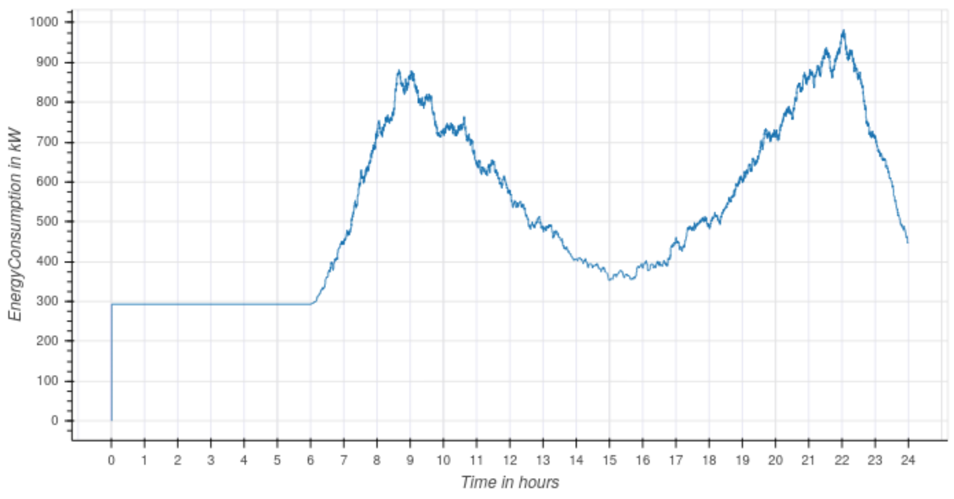


Figure 5.62.: *Energy Consumption of 1000 houses on working days*

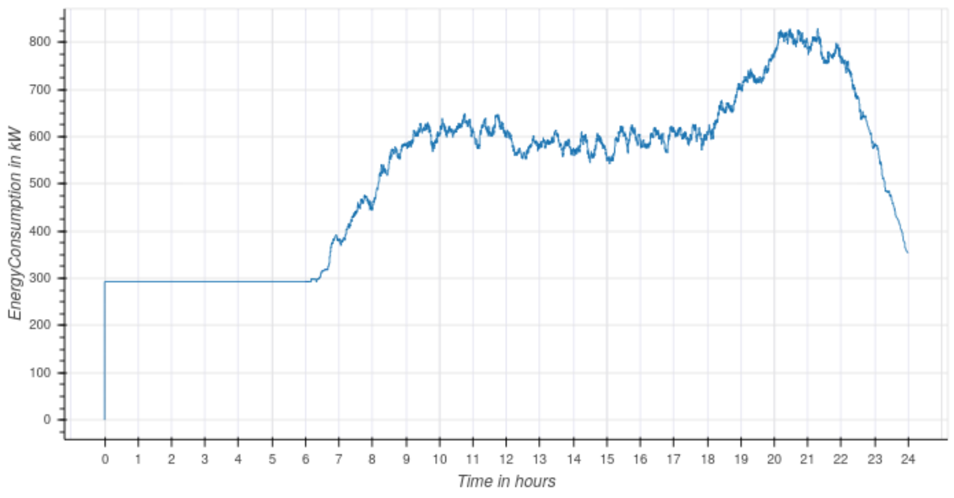


Figure 5.63.: *Energy Consumption of 1000 houses on none working days*

On days devoid of work obligations, energy consumption remains relatively steady between 10 a.m. and 6 p.m., as residents engage in various activities that involve heightened device usage. Significantly, the total energy consumption on these days surpasses that observed on regular working days. This disparity arises from the intensified utilization of electronic devices, as residents have more leisure time at their disposal compared to working days. The evening hours once again experience a surge in energy consumption, largely attributed to the augmented lighting requirements during this period.

Insights Collectively, these experiments provide comprehensive insights into the dynamic nature of energy consumption within the simulated houses. By differentiating between workdays and non-working days, the simulations underscore the role of human routines in shaping energy demand profiles. These observations not only facilitate a deeper understanding of residential energy usage but also serve as valuable inputs for future energy management strategies within smart home environments.

5.6.3. Software in the loop Evaluation

The assessment of computational overhead introduced by homomorphic encryption within the dataflow is a pivotal aspect of this evaluation. To gauge the impact on system performance, we conducted evaluations across three distinct scenarios: one without homomorphic encryption, one with partially homomorphic encryption, and one with fully homomorphic encryption. The primary objective was to compare their respective runtimes.

Experimental Setup

The simulations were executed within a virtual machine operating on Arch Linux, hosted on a Mac Pro workstation equipped with a 3.5 GHz 6-Core Intel Xeon E5 CPU, 16GB 1866 MHz DDR3 RAM, and an AMD FirePro D500 3072 MB GPU running MacOS Mojave. The virtual machine was configured to utilize 6 processor cores and 8192 MB of RAM. Both the encryption service and plaintext aggregation components were deployed on an external server, effectively eliminating topology-related discrepancies in the comparative analysis. This external server was equipped with a single 2.7 GHz Intel Xeon E5 Core. Communication between the simulation and the encryption service was facilitated via an Ethernet connection. This setup, involving the separation of the simulation and encryption service, served to mitigate any potential computational resource contention, ensuring that computational resources allocated to the simulation were not compromised by encryption service activities.

Encryption Service Implementation Currently, the encryption service employed open-source libraries that offer wrapper interfaces to their underlying low-level functions. These libraries seamlessly integrate encryption into the VICINITY architecture. For the fully homomorphic encryption aspect of the study, the HElib Library was utilized[108]. HElib implements the Brakerski-Gentry-Vaikuntanathan (BGV) scheme, incorporating numerous optimizations to enhance runtime efficiency. In the case of partially homomorphic encryption, the libhcs library[109] was integrated into the encryption service. This library encompasses a variety of partially homomorphic encryption schemes[109]. Our focus was on the implementation of the Paillier encryption scheme for additive homomorphic encryption, aligning with the specific requirements of our use case. Additionally, a straightforward Python script, based on the Python Flask framework[110], was developed. This script emulates the endpoints of the encryption service while operating on plain text input data.

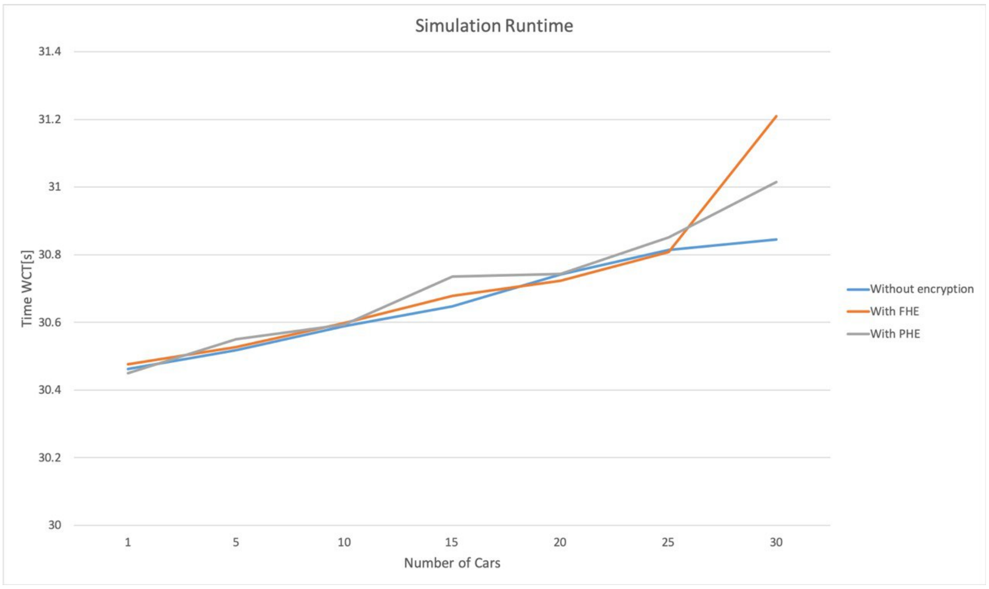


Figure 5.64.: *Runtime of the simulation with and without (fully, partial) homomorphic encryption*

Results

The runtime performance results of the various simulation runs, illustrated in Figure 5.64, demonstrate a notable upward trend with an increasing number of participating vehicles in the System under investigation. This observed increase in runtime is primarily attributed to the escalating computational demands incurred during the simulation of intricate communication processes among participants. The augmentation in runtime observed in simulation scenarios employing homomorphic encryption, in comparison to scenarios devoid of encryption, can be attributed to two distinct factors. Firstly, there is the computational overhead incurred on the service side due to the encryption process itself. Secondly, the extended time required for transmitting the substantial response ciphers, which are notably larger than their unencrypted counterparts, across the simulated network and the actual Ethernet infrastructure. It is imperative to acknowledge that these observed performance disparities may be influenced by the constraints imposed by the real-time scheduler employed for hardware-in-the-loop simulations. In situations characterized by a high volume of communications within the actual network or the transmission of exceedingly large payloads (such as an extensive number of ciphers), the real-time scheduler may need to selectively prioritize certain inbound or outbound communications to uphold its real-time capabilities. Beyond these considerations, the performance of our simulations was intrinsically contingent on the efficiency and congestion levels within the laboratory network. Notably, the state of the network had a discernible impact on the overall system performance. Additionally, the choice of cryptographic libraries played a pivotal role in shaping performance outcomes. For fully homomorphic encryption, we leveraged HElib [108], a library characterized by active development and ongoing optimization efforts. HElib implements the BGV encryption scheme, which undergoes constant refinement to leverage newly discovered improvements and performance enhancements. In contrast, our implementation of partially homomorphic encryption relied on libhcs [109]. While libhcs does offer support for cryptographic systems like Paillier, El-Gamal, and Damgard-Jurik, it's important to note that these schemes are no longer actively maintained, primarily due to the advent of practical fully homomorphic encryption solutions. Consequently, these older cryptographic schemes, lacking the capacity to capitalize on recent advancements, exhibited inferior performance in certain scenarios when juxtaposed with fully homomorphic encryption implementations.

Conclusions

One of the principal findings derived from this experiment is the observation that the overhead introduced by the utilization of homomorphic encryption, whether partially or fully homomorphic, constitutes only a fraction of the total runtime when compared to the runtime of our plaintext simulation. Notably, in the runtime comparison for ten cars, it is evident that fully homomorphic encryption performs nearly identically to plaintext. Even in scenarios involving 30 cars, where fully homomorphic encryption exhibits the highest runtime among the three approaches, the incurred overhead remains modest at approximately 1.2%. Given the considerable privacy enhancements offered by homomorphic encryption, this overhead can be deemed negligible for our application. These results hold significant value for our future applications. In practical use cases of homomorphic encryption within the VICINITY project, the number of simulated cars remains within realistic and feasible bounds. Consequently, we anticipate no significant runtime penalties attributable to the adoption of homomorphic encryption. However, this novel approach promises substantial improvements in the privacy of car owners, thereby enhancing the acceptance and appeal of the VICINITY project to them. Furthermore, this experiment affirms the applicability of both hardware-in-the-loop and software-in-the-loop simulations within the proposed simulation framework. It is important to acknowledge that the utilized real-time scheduler for simulation had an anticipated impact on the overall simulation runtime, primarily due to the necessity of interfacing with real hardware and aligning simulation time with real-time constraints. This impact, though expected, aligns seamlessly with our overarching approach of dynamically interchangeable models at runtime, which accommodate encapsulated simulators, providing a finer-grained representation of simulation time, and thereby enabling more detailed simulation steps. These findings serve as a foundational step in understanding the role of homomorphic encryption in simulation scenarios and the potential of hardware-in-the-loop and software-in-the-loop simulations within our framework. Future investigations may explore alternative real-time schedulers to further optimize performance in scenarios with extensive communications into real networks, mitigating potential limitations observed in this experiment.

Chapter 6

Conclusions and Outlook

6.1. Conclusion

In the pursuit of more efficient, secure, and effective cyber-physical energy systems (CPES), the integration of Internet of Things (IoT) infrastructures has emerged as a pivotal area of research. This dissertation, titled "Modeling and Simulation of Internet of Things Infrastructures for Cyber-Physical Energy Systems," delves into this domain, presenting a novel approach for model-based development and simulation-based verification and validation of IoT infrastructures within CPES. This research addresses the imperative need for early validation of system integration in CPES, a field that unites the realms of physical and cyber components to distribute energy efficiently and reliably.

6.1.1. Context and Significance

CPES represent a monumental leap in the energy sector by amalgamating energy generation, transmission, distribution, and consumption with digital communication and control technology. The physical components encompass the tangible elements of the energy system, while the cyber facets encompass the hardware, software, and communication technologies that enable the system to function seamlessly. In recent years, the rapid evolution of IoT platforms has created an extensive ecosystem comprising gateways, middlewares, and cloud platforms. Building smart devices and integrating them with value-added services necessitates careful platform selection. Nonetheless, the proliferation of diverse IoT network standards has complicated the replication of IoT systems. Ensuring the reliability and efficiency of IoT systems requires comprehensive testing. This entails the management of large-scale simulation nodes to replicate diverse communication scenarios, energy usage patterns, and decision-making processes. The need for precise modeling of every network component, including complex node interactions, underscores the intricacies involved in this domain.

6.1.2. Methodology and Contributions

The dissertation presents a methodological framework for the rapid prototyping of simulatable models for large and complex structures during the early phases of system analysis and conceptual design. This approach provides engineers with a powerful toolset to effectively model IoT systems while considering the power and energy requirements of future intelligent cyber-physical energy systems. The contributions of this research can be distilled into three key facets:

1. **Model-Based Approach**: The development of a model-based approach tailored for the early conceptual design phase of IoT systems stands as a significant contribution. The inherent complexity of IoT systems necessitates specialized simulation and modeling techniques, capable of handling their intricacies. This approach offers a path to navigate this complexity and facilitates the development of IoT systems. This contribution answers the **research questions 1 and 2** from chapter 1.4.1. The details of the developed MBSE approach and it's application are clearly shown in sections 5.2 and 5.3, where the modeling languages SysML and SysMD can unfold to their full potential. The complexity of the use case in this dissertation is through the use of these techniques still manageable.

2. **Simulation and Emulation Framework**: The creation of a simulation and emulation framework for IoT infrastructures (refer to chapter 4), particularly attuned to the power and energy requirements of future intelligent cyber-physical energy systems, represents another substantial contribution. This framework encompasses a rapid prototyping method for simulatable models during the initial stages of system analysis and conceptual system design. This approach empowers IoT engineers to develop and optimize systems while meticulously considering energy and power aspects. The Framework clearly answers **research question 3** from section 1.4.1. In chapter 5, the simulation framework shows it's capabilities to efficiently simulate IoT infrastructures in order to address their power and energy usage.

3. **Integration and Application**: The integration and application of the developed methodology and simulator in a real-world scenario within the VICINITY project adds practical relevance to this research. The VICINITY project, focused on creating a virtual neighborhood platform for IoT devices, poses unique challenges related to energy and power aspects. The simulator's ability to handle complex infrastructures, including energy network dimensioning for smart neighborhoods, underscores its efficacy in addressing the distinctive requirements of future intelligent cyber-physical energy systems. In the use case of this dissertation (chapter 5) the developed simulation framework is seamlessly integrated into the IoT scenarios arising from the VICINITY project. This clearly answers **research question 4** of the target elicitation in chapter 1.4.1.

6.1.3. Thesis Organization and Contributions

The dissertation's organization reflects its comprehensive approach to integrating IoT infrastructures into cyber-physical energy systems. It commences with an in-depth exploration of state-of-the-art IoT simulators and their computational models. This investigation serves as a foundation for the development of a specialized simulator tailored to the unique demands of the VICINITY project.

The central premise of this research is the presentation of a holistic development process for IoT infrastructures within cyber-physical energy systems. The approach leverages model-based systems engineering techniques, adapted to the specific challenges of IoT development. The dissertation illustrates a model-based development approach aimed at the early conceptual design phase of IoT systems. This approach acknowledges the elevated complexity of such systems and offers a means to navigate it effectively.

A comprehensive case study further solidifies the methodology and the simulator's utility. The scenario is meticulously modeled using SysML and subsequently translated for use within the simulator. Experimental evaluations encompassing hardware in the loop, software in the loop, and human in the loop validate the effectiveness of this approach.

6.1.4. In Conclusion

In summary, this dissertation introduces a pioneering approach to seamlessly integrate IoT infrastructures into the intricate tapestry of cyber-physical energy systems. The proposed methodology targets the early phases of IoT development, including system analysis and conceptual design, with an emphasis on the rapid prototyping of simulatable models. By harnessing SysML models to faithfully represent system behavior and address the formidable complexity of IoT systems, this research equips engineers with the tools needed to usher in a new era of energy efficiency and reliability.

The developed simulator, battle-tested in the VICINITY project and demonstrated to address the unique requirements of power and energy usage in future intelligent cyber-physical energy systems, stands as a testament to the practical implications of this research. These contributions, taken together, hold the promise of steering IoT systems toward a future where energy and power requirements are not just met but optimized. In this vision, IoT seamlessly empowers the intelligent, efficient, and secure distribution of energy, forging a path toward a sustainable and technologically advanced future.

6.2. Summary

The dissertation dives into a transformative realm where the Internet of Things (IoT) intersects with cyber-physical energy systems (CPES). CPES, an evolution in energy management, fuses physical components with digital technology. In this paradigm shift, IoT plays a pivotal role, demanding advanced modeling and simulation methodologies to ensure seamless integration and optimize energy systems.

The journey begins with a comprehensive examination of IoT platforms. These platforms form a complex ecosystem encompassing gateways, middlewares, and cloud platforms. The research highlights the intricacies of selecting the right platforms when connecting smart devices and creating value-added services. Moreover, it delves into the daunting challenge of dealing with numerous IoT network standards, emphasizing the need for replicable IoT systems.

A cornerstone of this research lies in the testing of IoT systems. Replicating scenarios involving communication, energy consumption, and decision-making processes demands the ability to manage a multitude of simulation nodes. The simulator's complexity also extends to handling intricate interactions between these nodes.

To address the multifaceted nature of IoT systems, the dissertation introduces a model-based methodology. This innovative approach enables the rapid prototyping of simulatable models, empowering engineers to navigate IoT's complexities effectively. It caters to the early phases of IoT system development, specifically system analysis and conceptual system design.

A pivotal contribution of this research is the development of a simulation and emulation framework tailored to the energy requirements of future intelligent CPES. This framework facilitates the rapid prototyping of simulatable models, ideal for handling complex IoT systems. It aims to support engineers in developing systems that consider energy and power aspects from the outset.

The research takes a leap from theory to practice by integrating the developed methodology and simulator into the VICINITY project, a real-world endeavor. The VICINITY project endeavors to create a virtual neighborhood platform for IoT devices. This dissertation provides a novel strategy for supporting the development of specialized IoT infrastructures crucial to VICINITY, particularly those related to energy and power aspects. The simulator's application to complex infrastructures contributes to the dimensioning of energy networks in future smart neighborhoods.

In summary, this dissertation pioneers the integration of IoT infrastructures into the realm of CPES. The model-based methodology offers a structured approach for navigating the intricacies of IoT systems. The simulation framework, fine-tuned for energy systems, prioritizes energy and power considerations, ensuring efficient energy distribution. Through practical validation in the VICINITY project, IoT emerges as the linchpin for intelligent, sustainable energy systems. This research lays the foundation for a future where energy systems are optimized, sustainable, and powered by the Internet of Things.

6.3. Future Work and Outlook

6.3.1. Simulation Approach

As previously delineated in [87], the current prototype supports the utilization of continuous-time models, particularly through hybrid models. However, the continuous interaction among such models can present challenges concerning the discrete-time architecture of the simulator. A proposed solution lies within the realm of the generalized DEVS specification [111], which employs polynomial events to approximate continuous output.

Large-scale IoT scenarios stand to benefit significantly from parallel and distributed simulation techniques. By integrating MPI, OMNeT++ already offers support for parallel and distributed execution. Future endeavors should harness this existing architecture to enable the developed simulator to fully leverage OMNeT++ capabilities.

The versatile simulation framework proposed herein can extend its utility to evaluate other software or hardware solutions tailored for the Internet of Things. For instance, the approach detailed in [112] can undergo evaluation for performance and design within this framework.

The functional mock-up interface (FMI) emerges as pivotal for simulating Cyber-Physical Systems (CPS). To further enhance integration at the lowest level of the framework, FMI integration into OMNeT++ is imperative. While SystemC models have already been incorporated into the simulation without FMI, precision-driven simulation outcomes can greatly benefit from FMI integration, alongside languages like Modelica.

Lastly, runtime evaluations have affirmed the feasibility of homomorphic encryption for practical implementations. In forthcoming work, we intend to integrate a fully homomorphic micro-service, as illustrated in Figure 5.17, into the VICINITY architecture. Our findings indicate that this implementation introduces negligible runtime overhead while offering substantial advantages in terms of privacy compared to plaintext scenarios and versatility compared to partially homomorphic encryption. Leveraging the VICINITY Bridge Feature (see 5.2.2), this approach can undergo comprehensive testing in a simulated environment, devoid of potential disruptions to real-world applications.

6.3.2. SysMD

As this dissertation has shown, in the context of increasingly complex products and systems across various domains, the need for advanced modeling tools like SysML has become crucial. However, this poses a challenge as many systems modeling experts lack domain-specific knowledge, resulting in potential issues. Additionally, hiring or training modeling experts can be costly and time-consuming.

To address these challenges and provide a high-level abstraction for systems modeling and knowledge representation, a framework and modelling language called SysMD is currently under development. This framework primarily focuses on modeling requirements, structure, and constraints, offering several key features:

1. **Integration of Documentation and Models**: The framework seamlessly combines modeling with Markdown documents, enhancing the maintainability of knowledge bases. It allows for the inclusion of formatted text, tables, and references to external documents.

2. **Inclusive Language**: The framework employs a syntax designed to be accessible to individuals without prior modeling expertise, making it inclusive for various stakeholders, including domain and management experts.

3. **Continuous Consistency Checking**: To support formal analysis, the framework incorporates constraint propagation techniques, facilitating quick evaluations of design element dependencies—a valuable asset, particularly in agile development processes.

In terms of modeling languages, it's worth noting that existing options like UML, SysML, and OWL have limitations, especially concerning human readability and formal analysis capabilities.

The framework's target audience includes domain experts who may not possess modeling experience. To address this, the framework relies on natural language statements, predefined semantic relationships such as ”isA” and ”hasA,” and encourages documentation as an integral part of the modeling process. Additionally, it offers extensibility for specialized use cases.

In summary, the framework presents a user-friendly, inclusive, and highly adaptable approach to systems modeling and knowledge representation. Its development aligns with the growing need for accessible modeling tools in complex domains, which may have implications for future work, especially in the realm of simulation frameworks.

Bibliography

[1] *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2023, with forecasts from 2022 to 2030.* `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`.

[2] Aida Mynzhasova, Carna Radojicic, Christopher Heinz, Johannes Kölsch, Christoph Grimm, Juan Rico, Keith Dickerson, Raúl García-Castro, and Victor Oravec. “Drivers, standards and platforms for the IoT: Towards a digital VICINITY”. In: *2017 Intelligent Systems Conference (IntelliSys).* IEEE. 2017, pp. 170–176.

[3] *M2M World of Connected Services.* `https://www.iot-now.com/wp-content/uploads/2013/07/M2M_world_of_connected_serv.jpg`.

[4] *The Fourth Industrial Revolution and the Probable Future of Learning.* `https://imphalreviews.in/the-fourth-industrial-revolution-and-the-probable-future-of-learning/`.

[5] Reinhard Haberfellner, Olivier de Weck, Ernst Fricke, and Siegfried Vössner. *Systems Engineering: Grundlagen und Anwendung.* 2012.

[6] Johannes Kölsch, Carna Zivkovic, Yajuan Guan, and Christoph Grimm. “An Introduction to the Internet of Things”. In: *IoT Platforms, Use Cases, Privacy, and Business Models.* Springer, Cham, 2020, pp. 1–19.

[7] Johannes Koch and Christoph Grimm. “Cyber-Physikalische Systeme”. In: *Handbuch Digitalisierung* (2022), p. 315.

[8] *The Internet Gopher Protocol (a distributed document search and retrieval protocol).* `https://tools.ietf.org/html/rfc1436`.

[9] J. Postel. *Transmission Control Protocol.* RFC 793. RFC Editor, Sept. 1981. URL: `https://www.rfc-editor.org/rfc/rfc793.txt`.

[10] Berners-Lee Tim and Cailliau Robert. *WorldWideWeb: Proposal for a HyperText Project.* 1990. URL: `https://www.w3.org/Proposal.html`.

[11] *The little-known story of the first IoT device.* `https://www.ibm.com/blogs/industries/little-known-story-first-iot-device/`.

[12] Mark Weiser. “The computer for the 21st century”. In: *Scientific American* (Sept. 1991).

[13] Kevin Ashton. “That Internet of Things Thing”. In: *RFID Journal* (2009).

[14] International Telecommunication Union. “Overview of the Internet of Things”. In: *Recommendation ITU-T Y.2060* (2012).

[15] *The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast.* `https://www.idc.com/getdoc.jsp?containerId=prUS45213219`.

[16] Schelbey Z., Hartke K., and Bormann C. “RFC 7252 - The Constrained Application Protocol (CoAP)”. In: *Internet Engineering Task Force (IETF).* 2014.

[17] J. Postel. *DoD standard Internet Protocol.* RFC 760. RFC Editor, Jan. 1980. URL: `https://www.rfc-editor.org/rfc/rfc760.txt`.

[18] Henrik Frystyk Nielsen. “The Hypertext Transfer Protocol in the World-Wide Web”. twitter.com/frystyk. CERN, Geneva, Switzerland: Aalborg University, Denmark, Sept. 1994.

[19] J. Postel. *User Datagram Protocol.* RFC 768. RFC Editor, Aug. 1980. URL: `https://www.rfc-editor.org/rfc/rfc768.txt`.

[20] Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. RFC Editor, Sept. 1999. URL: `https://tools.ietf.org/html/rfc2616`.

[21] Roy Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. University of California, Irvine, Jan. 2000.

[22] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON).* RFC 4627. RFC Editor, July 2006. URL: `https://tools.ietf.org/html/rfc4627`.

[23] Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu, and Weijun Qin. “IOT Gateway: Bridging Wireless Sensor Networks into Internet of Things”. In: *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing.* 2010, pp. 347–352.

[24] Hao Chen, Xueqin Jia, and Heng Li. “A brief introduction to IoT gateway”. In: *Communication Technology and Application (ICCTA 2011), IET International Conference on.* 2011, pp. 610–613.

[25] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. “Internet of Things: A Survey of Enabling Technologies, Protocols and Applications”. In: *IEEE COMMUNICATION SURVEYS & TUTORIALS* 17.4 (2015), pp. 2347–2376.

[26] B.N. Silva, M. Khan, and Han K. “Internet of Things: A Comprehensive Review of Enabling Technologies, Architecture, and Challenges”. In: *IETE Technical Review* (2017), pp. 1–16.

[27] D. Minoli, K. Sohraby, and B. Occhiogrosso. “IoT Considerations, Requirements, and Architectures for Smart Buildings-Energy Optimization and Next-Generation Building Management Systems”. In: *IEEE Internet of Things Journal* 4.1 (Feb. 2017), pp. 269–283. DOI: `10.1109/JIOT.2017.2647881`.

[28] Yajuan Guan, Wei Feng, Emilio J. Palacios-Garcia, Juan C. Vásquez, and Josep M. Guerrero. “VICINITY Platform-based Load Scheduling Method by Considering Smart Parking and Smart Appliance”. In: *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. IEEE. 2019.

[29] Rahul Dagar, Subhranil Som, and Sunil Kumar Khatri. “Smart Farming - IoT in Agriculture”. In: *Proceedings of the International Conference on Inventive Research in Computing Applications (ICIRCA 2018)*. 2018.

[30] Meonghun Lee, Jeonghwan Hwang, and Hyun Yoe. “Agricultural Production System based on IoT”. In: *2013 IEEE 16th International Conference on Computational Science and Engineering*. IEEE. 2013.

[31] McAfee. *Key Findings from our Survey on Identity Theft, Family Safety and Home Network Security*. `https://www.mcafee.com/blogs/consumer/key-findings-from-our-survey-on-identity-theft-family-safety-and-home-network-security/`. 2018.

[32] Marie Madeleine Uwiringiyimana, Gomathi Nandagopal, Yajuan Guan, Sašo Vinkovič, Johannes Kölsch, and Christopher Heinz. “IoT Platforms”. In: *IoT Platforms, Use Cases, Privacy, and Business Models*. Springer, Cham, 2020, pp. 21–49.

[33] *Hisense*. `http://global.hisense.com/`. (Visited on 11/05/2019).

[34] Taein Kwon, Eunjeong Park, and Hyukjae Chang. “Smart Refrigerator for Healthcare Using Food Image Classification”. en. In: *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics - BCB ’16*. Seattle, WA, USA: ACM Press, 2016, pp. 483–484. DOI: `10.1145/2975167.2985644`. URL: `http://dl.acm.org/citation.cfm?doid=2975167.2985644`.

[35] Agus Kurniawan. *Getting Started with Windows 10 IoT Core for Raspberry Pi 3*. PE Press, 2016.

[36] Ruediger Follmann and Tony Zhang. *Banana Pro Blueprints*. Packt Publishing Ltd, 2015.

[37] *PINE A64 (+)*. en-US. `https://www.pine64.org/devices/single-board-computers/pine-a64/`. (Visited on 10/11/2019).

[38] *Cubietruck — CubieBoard*. en-US. `http://cubieboard.org/tag/cubietruck/`. (Visited on 10/11/2019).

[39] *Arduino - IntelEdison*. `https://www.arduino.cc/en/ArduinoCertified/IntelEdison`. (Visited on 10/11/2019).

[40] Keith Dickerson, Christopher Heinz, Raúl García-Castro, Flemming Sveen, and etc. *Analysis of Standardisation Context and Recommendations for Standards Involvement*. 2016. URL: `https://vicinity2020.eu/vicinity/sites/default/files/documents/vicinity_d2.1_analysis_of_standardisation_context_and_recommendations_for_standards_involvement.pdf`.

[41] *IoT Toolkit Overview*. `https://docs.devicehive.com/v2.0/docs/iot-toolkit-overview`. (Visited on 10/17/2019).

[42] Massimo Villari, Antonio Celesti, Maria Fazio, and Antonio Puliafito. “AllJoyn Lambda: An architecture for the management of smart environments in IoT”. In: *2014 International Conference on Smart Computing Workshops*. IEEE. 2014, pp. 9–14.

[43] *IoTivity*. `https://iotivity.org/about`.

[44] *Kura Wires overview*. `https://eclipse.github.io/kura/wires/kura-wires-intro.html`. (Visited on 11/12/2019).

[45] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. “Middleware for Internet of Things: A Survey”. In: *IEEE Internet of Things Journal* 3.1 (Feb. 2016), pp. 70–95. ISSN: 2372-2541. DOI: `10.1109/JIOT.2015.2498900`.

[46] Sergiy Nikitin and Minna Lappalainen. *Tekes project proposal: SOFIA Full title: Seamless Operation of Forest Industry Applications*. 2010.

[47] *Vicinity — Open virtual neighbourhood network to connect IoT infrastructures and smart objects*. `https://vicinity2020.eu/vicinity/`. (Visited on 10/23/2019).

[48] Bhumi Nakhuva and Tushar Champaneria. “Study of various internet of things platforms”. In: *International Journal of Computer Science & Engineering Survey* 6.6 (2015), pp. 61–74.

[49] *AWS IoT Core Features - Amazon Web Services*. `https://www.amazonaws.cn/en/iot-core/features/`. (Visited on 10/23/2019).

[50] *Microsoft Azure Tutorial for Beginners: Learn in 1 Day*. `https://www.guru99.com/microsoft-azure-tutorial.html`. (Visited on 10/23/2019).

[51] *Google Cloud Platform*. `https://www.conceptdraw.com/solution-park/computer-networks-google-cloud-platform`. (Visited on 10/29/2019).

[52] Grant and Svetlana. *GSMA White Paper: 3GPP Low Power Wide Area Technologies*. `https://www.gsma.com/iot/wp-content/uploads/2016/10/3GPP-Low-Power-Wide-Area-Technologies-GSMA-White-Paper.pdf`. 2016.

[53] VINKOVIČ Sašo and OJSTERŠEK Milan. “The internet of things communication protocol for devices with low memory footprint”. In: *International journal of ad hoc and ubiquitous computing* 24.4 (2017), pp. 271–281.

[54] Raúl García Castro (leading author). *VICINITY D2.2: Detailed Specification of the Semantic Model*. Tech. rep. UPM, 2017. URL: `https://www.vicinity2020.eu/vicinity/sites/default/files/documents/vicinity_d2.2_vicinitysemanticmodel_v1.0.pdf`.

[55] Viktor Oravec (leading author). *D1.6: VICINITY Architectural Design.* Tech. rep. BVR, 2017. URL: `https://www.vicinity2020.eu/vicinity/content/d16-vicinityd16architecturaldesign10`.

[56] Craig Gentry. "A fully homomorphic encryption scheme". PhD thesis. https://crypto.stanford.edu/craig: Stanford University, 2009.

[57] Craig Gentry. "Computing Arbitrary Functions of Encrypted Data". In: *Commun. ACM* 53.3 (2010), pp. 97–105.

[58] A. Gluhak, S. Krco, M. Nati, D. Pfisterer, N. Mitton, and T. Razafindralambo. "A survey on facilities for experimental internet of things research". In: *IEEE Communications Magazine* 49.11 (Nov. 2011), pp. 58–67. DOI: `10.1109/MCOM.2011.6069710`.

[59] C. Fehling et al. *Kompakt-Lexikon Wirtschaftsinformatik.* 2nd ed. International series of monographs on physics. Springer Fachmedien Wiesbaden, 2013. ISBN: 978-3-658-03028-5.

[60] Gabriel A Wainer. *Discrete-event modeling and simulation: a practitioner's approach.* CRC press, 2009.

[61] Dave Evans. *The Internet of Things How the Next Evolution of the Internet Is Changing Everything.* Tech. rep. Cisco Internet Business Solutions Group (IBSG), 2011. URL: `https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`.

[62] Stefano Ferretti and Gabriele D'Angelo. "Smart Shires: The Revenge of Countrysides". In: *Proceedings of the IEEE Symposium on Computers and Communications.* ISCC '16. Washington, DC, USA: IEEE Computer Society, 2016. DOI: `10.1109/ISCC.2016.7543862`.

[63] Gabriele D'Angelo, Stefano Ferretti, and Vittorio Ghini. "Multi-level simulation of Internet of Things on smart territories". In: *Simulation Modelling Practice and Theory (SIMPAT)* 73 (2017), pp. 3–21. ISSN: 1569-190X. DOI: `10.1016/j.simpat.2016.10.008`. URL: `http://www.sciencedirect.com/science/article/pii/S1569190X16302507`.

[64] Gabriele D'Angelo, Stefano Ferretti, and Vittorio Ghini. "Distributed Hybrid Simulation of the Internet of Things and Smart Territories". In: *To appear in Concurrency and Computation: Practice and Experience* 30.9 (2018). ISSN: 1532-0634. DOI: `10.1002/cpe.4370`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4370`.

[65] Tony Markel, Aaron Brooker, T Hendricks, V Johnson, Kenneth Kelly, Bill Kramer, Michael O'Keefe, Sam Sprik, and Keith Wipke. "ADVISOR: a systems analysis tool for advanced vehicle modeling". In: *Journal of power sources* 110.2 (2002), pp. 255–266.

[66] Giancarlo Fortino, Wilma Russo, and Claudio Savaglio. "Agent-oriented modeling and simulation of IoT networks". In: *2016 federated conference on computer science and information systems (FedCSIS).* IEEE. 2016, pp. 1449–1452.

[67] Giancarlo Fortino, Wilma Russo, and Claudio Savaglio. "Simulation of Agent-oriented Internet of Things Systems." In: *WOA*. 2016, pp. 8–13.

[68] G. Fortino, R. Gravina, W. Russo, and C. Savaglio. "Modeling and Simulating Internet-of-Things Systems: A Hybrid Agent-Oriented Approach". In: *Computing in Science Engineering* 19.5 (2017), pp. 68–76. ISSN: 1521-9615. DOI: `10.1109/MCSE.2017.3421541`.

[69] Vilen Looga, Zhonghong Ou, Yang Deng, and Antti Ylä-Jääski. "Mammoth: A massive-scale emulation platform for internet of things". In: *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*. Vol. 3. IEEE. 2012, pp. 1235–1239.

[70] *NS2 Wiki*. http://nsnam.sourceforge.net/wiki/index.php/Main_Page.

[71] *NS3 Wiki*. https://www.nsnam.org/wiki/index.php/Main_Page.

[72] *PDNS*. https://www.cc.gatech.edu/computing/compass/pdns/.

[73] *GTNetS*. http://griley.ece.gatech.edu/MANIACS/GTNetS/.

[74] A Sobeih, J C Hou, Lu-Chuan Kung, Ning Li, Honghai Zhang, Wei-Peng Chen, Hung-Ying Tyan, and Hyuk Lim. "J-Sim: a simulation and emulation environment for wireless sensor networks". In: *IEEE Wireless Communications* 13.4 (2006), pp. 104–119. ISSN: 1536-1284. DOI: `10.1109/MWC.2006.1678171`.

[75] Tronje Krop, Michael Bredel, Matthias Hollick, and Ralf Steinmetz. "JiST/MobNet: Combined Simulation, Emulation, and Real-world Testbed for Ad Hoc Networks". In: *Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*. WinTECH '07. New York, NY, USA: ACM, 2007, pp. 27–34. ISBN: 978-1-59593-738-4. DOI: `10.1145/1287767.1287774`. URL: `http://doi.acm.org/10.1145/1287767.1287774`.

[76] *Contiki: The Open Source Operating System for the Internet of Things*. http://www.contiki-os.org/.

[77] Stanford Information Networks Group. *TinyOS Wiki, TOSSIM*. URL: `http://tinyos.stanford.edu/tinyos-wiki/index.php?title=TOSSIM`.

[78] Jawwad Shamsi and Monica Brockmeyer. "DSSimulator: Achieving million node simulation of Distributed Systems". In: *Spring Simulation Multiconference*. 2005.

[79] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. "GloMoSim: a library for parallel simulation of large-scale wireless networks". In: *Proceedings. Twelfth Workshop on Parallel and Distributed Simulation PADS'98 (Cat. No. 98TB100233)*. IEEE. 1998, pp. 154–161.

[80] *Omnet++ Simulation Manual*. URL: `https://omnetpp.org/doc/omnetpp/manual/`.

[81] Sung Park, Andreas Savvides, and Mani B Srivastava. “SensorSim: A Simulation Framework for Sensor Networks”. In: *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems.* MSWIM ’00. New York, NY, USA: ACM, 2000, pp. 104–111. ISBN: 1-58113-304-9. DOI: `10.1145/346855.346870`. URL: `http://doi.acm.org/10.1145/346855.346870`.

[82] Gilbert Chen, Joel Branch, Michael Pflug, Lijuan Zhu, and Boleslaw Szymanski. “SENSE: a wireless sensor network simulator”. In: *Advances in pervasive computing and networking.* Springer, 2005, pp. 249–267.

[83] *Emulab.* https://www.emulab.net/.

[84] *ATEMU.* http://www.hynet.umd.edu/research/atemu/.

[85] Giacomo Brambilla, Marco Picone, Simone Cirani, Michele Amoretti, and Francesco Zanichelli. “A Simulation Platform for Large-scale Internet of Things Scenarios in Urban Environments”. In: *Proceedings of the First International Conference on IoT in Urban Space.* URB-IOT ’14. Rome, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 50–55. ISBN: 978-1-63190-037-2. DOI: `10.4108/icst.urb-iot.2014.257268`. URL: `http://dx.doi.org/10.4108/icst.urb-iot.2014.257268`.

[86] Johannes Kölsch, Christopher Heinz, Sebastian Schumb, and Christoph Grimm. “Hardware-in-the-loop simulation for Internet of Things scenarios”. In: *2018 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES).* IEEE. 2018, pp. 1–6.

[87] Johannes Kölsch, Axel Ratzke, and Christoph Grimm. “Co-Simulating the Internet of Things in a Smart Grid use case scenario”. In: *2019 7th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES).* IEEE. 2019, pp. 1–6.

[88] Johannes Kölsch, Axel Ratzke, Christoph Grimm, Christopher Heinz, and Gomathi Nandagopal. “Simulation based validation of a Smart Energy Use Case with Homomorphic Encryption”. In: *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS).* IEEE. 2019, pp. 255–262.

[89] Johannes Kölsch, Christopher Heinz, Axel Ratzke, and Christoph Grimm. “Simulation-Based Performance Validation of Homomorphic Encryption Algorithms in the Internet of Things”. In: *Future Internet* 11.10 (2019), p. 218.

[90] Johannes Kölsch, Sebastian Post, Carna Zivkovic, Axel Ratzke, and Christoph Grimm. “Model-based development of smart home scenarios for IoT simulation”. In: *2020 8th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems.* IEEE. 2020, pp. 1–6.

[91] B. Zeigler. *Theory of modeling and simulation. 1st.* International series of monographs on physics. 1976. ISBN: 9780198520115.

[92] Hans Vangheluwe. “The discrete event system specification (DEVS) Formalism”. In: *Course Notes, Course: Modeling and Simulation, McGill University, Montreal Canada* 13 (2001).

[93] James J Nutaro. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Wiley Publishing, 2010.

[94] Katie Costello and Sarah Cornelia Hippold. *Gartner Says Worldwide Sales of Smartphones Returned to Growth in First Quarter of 2018*. Accessed: 01-22-2020. URL: `https://www.gartner.com/en/newsroom/press-releases/2018-05-29-gartner-says-worldwide-sales-of-smartphones-returned-to-growth-in-first-quarter-of-2018`.

[95] Klaus Wehrle, Mesut Günes, and James Gross. “OMNeT++”. In: *Modeling and Tools for Network Simulation*. Springer Berlin Heidelberg, 2010, pp. 35–59.

[96] Pascal Paillier. “Public-key Cryptosystems Based on Composite Degree Residuosity Classes”. In: *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT’99. Prague, Czech Republic: Springer-Verlag, 1999, pp. 223–238. ISBN: 3-540-65889-0. URL: `http://dl.acm.org/citation.cfm?id=1756123.1756146`.

[97] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Report 2011/277. 2011.

[98] “Total number of reported fires in the United States in 2018, by type”. In: *Fire loss in the United States during 2018*. National Fire Protection Association, Sept. 2019.

[99] “Number of reported burglary cases in the United States from 1990 to 2018”. In: *Crime in the United States 2018*. FBI, Sept. 2019.

[100] Antje Gößwald, Anja Schienkiewitz, Enno Nowossadeck, and Markus Busch. “Prävalenz von Herzinfarkt und koronarer Herzkrankheit bei Erwachsenen im Alter von 40 bis 79 Jahren in Deutschland”. In: *Bundesgesundheitsblatt - Gesundheitsforschung - Gesundheitsschutz*. Vol. 56. 5/6. Robert Koch-Institut, Epidemiologie und Gesundheitsberichterstattung, 2013. DOI: `10.1007/s00103-013-1666-9`.

[101] Johannes Koch, Alexander Wansch, and Christoph Grimm. “Knowledge modeling of power grids with SysMD”. In: *2022 10th Workshop on Modelling and Simulation of Cyber-Physical Energy Systems (MSCPES)*. IEEE. 2022, pp. 1–6.

[102] *VICINITY Get Started Documentation - Release 0.6.2*. bAvenir, Bratislava, Slovakia, 2019.

[103] Apple Inc. *About App Development with UIKit*. Accessed: 01-22-2020. URL: `https://developer.apple.com/documentation/uikit/about_app_development_with_uikit`.

[104] A. Syromiatnikov and D. Weyns. “A Journey through the Land of Model-View-Design Patterns”. In: *2014 IEEE/IFIP Conference on Software Architecture*. Apr. 2014, pp. 21–30. DOI: `10.1109/WICSA.2014.13`.

[105] Apple Inc. *Human Interface Guidelines*. Accessed: 01-22-2020. URL: `https://developer.apple.com/design/human-interface-guidelines`.

[106] *Thing Descriptions for integrators of IoT infrastructures and Value-added services*. Accessed: 01-22-2020. URL: `https://github.com/vicinityh2020/vicinity-agent/blob/master/docs/TD.md`.

[107] S. Kaebisch et al. *Web of Things (WoT) Thing Description*. Tech. rep. World Wide Web Consortium (W3C), Nov. 2019.

[108] Shai Halevi and Victor Shoup. *HElib - Homomorphic-Encryption Library*. https://github.com/shaih/HElib.

[109] Marc Tiehuis. *libhcs*. https://github.com/tiehuis/libhcs.

[110] Armin Ronacher. *Python Flask*. https://palletsprojects.com/p/flask/.

[111] N. Giambiasi, B. Escude, and S. Ghosh. “GDEVS: a generalized discrete event specification for accurate modeling of dynamic systems”. In: *Proceedings 5th International Symposium on Autonomous Decentralized Systems*. Mar. 2001, pp. 464–469. DOI: `10.1109/ISADS.2001.917452`.

[112] Hyunbum Kim and Jalel Ben-Othman. “A Collision-Free Surveillance System Using Smart UAVs in Multi Domain IoT”. In: *IEEE Communications Letters* 22.12 (2018), pp. 2587–2590.

Appendix A

SysML Diagrams

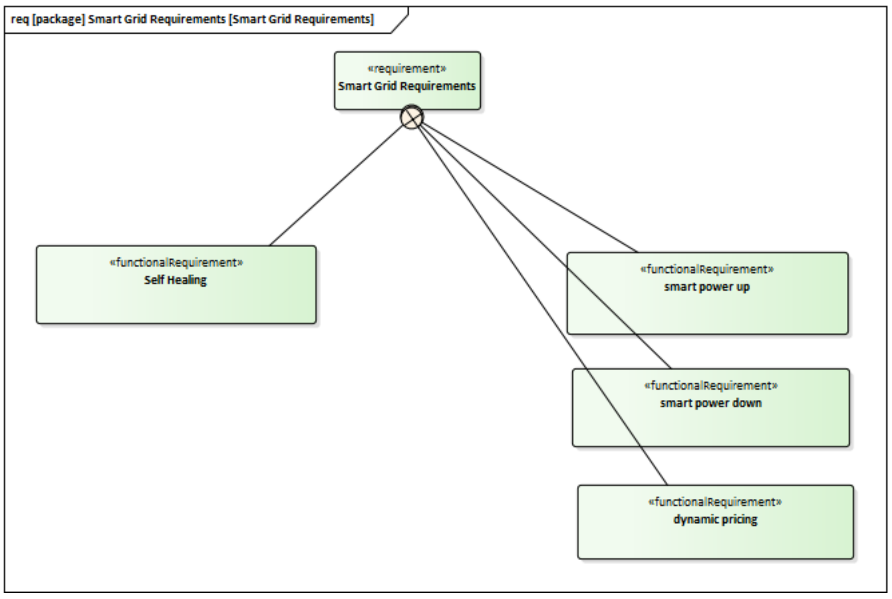


Figure A.1.: *Smart Grid Requirement Diagram*

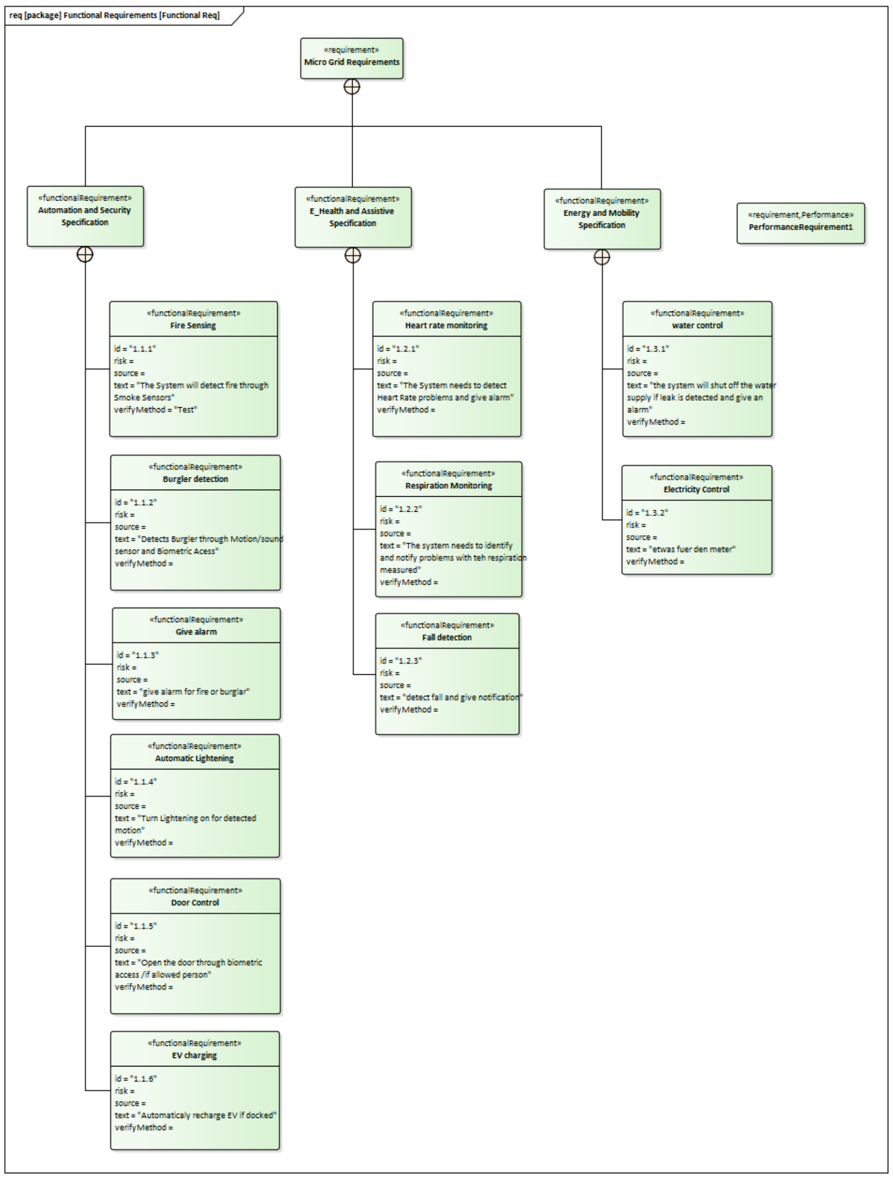

req [package] Functional Requirements [Functional Req]
«requirement» Micro Grid Requirements
«functionalRequirement» Automation and Security Specification
«functionalRequirement» E_Health and Assistive Specification
«functionalRequirement» Energy and Mobility Specification
«requirement,Performance» PerformanceRequirement1
«functionalRequirement» Fire Sensing
id = "1.1.1"
risk =
source =
text = "The System will detect fire through Smoke Sensors"
verifyMethod = "Test"
«functionalRequirement» Burgler detection
id = "1.1.2"
risk =
source =
text = "Detects Burgler through Motion/sound sensor and Biometric Acess"
verifyMethod =
«functionalRequirement» Give alarm
id = "1.1.3"
risk =
source =
text = "give alarm for fire or burglar"
verifyMethod =
«functionalRequirement» Automatic Lightening
id = "1.1.4"
risk =
source =
text = "Turn Lightening on for detected motion"
verifyMethod =
«functionalRequirement» Door Control
id = "1.1.5"
risk =
source =
text = "Open the door through biometric access /if allowed person"
verifyMethod =
«functionalRequirement» EV charging
id = "1.1.6"
risk =
source =
text = "Automaticaly recharge EV if docked"
verifyMethod =
«functionalRequirement» Heart rate monitoring
id = "1.2.1"
risk =
source =
text = "The System needs to detect Heart Rate problems and give alarm"
verifyMethod =
«functionalRequirement» Respiration Monitoring
id = "1.2.2"
risk =
source =
text = "The system needs to identify and notify problems with teh respiration measured"
verifyMethod =
«functionalRequirement» Fall detection
id = "1.2.3"
risk =
source =
text = "detect fall and give notification"
verifyMethod =
«functionalRequirement» water control
id = "1.3.1"
risk =
source =
text = "the system will shut off the water supply if leak is detected and give an alarm"
verifyMethod =
«functionalRequirement» Electricity Control
id = "1.3.2"
risk =
source =
text = "etwas fuer den meter"
verifyMethod =

Figure A.2.: *Micro Grid Requirement Diagram*

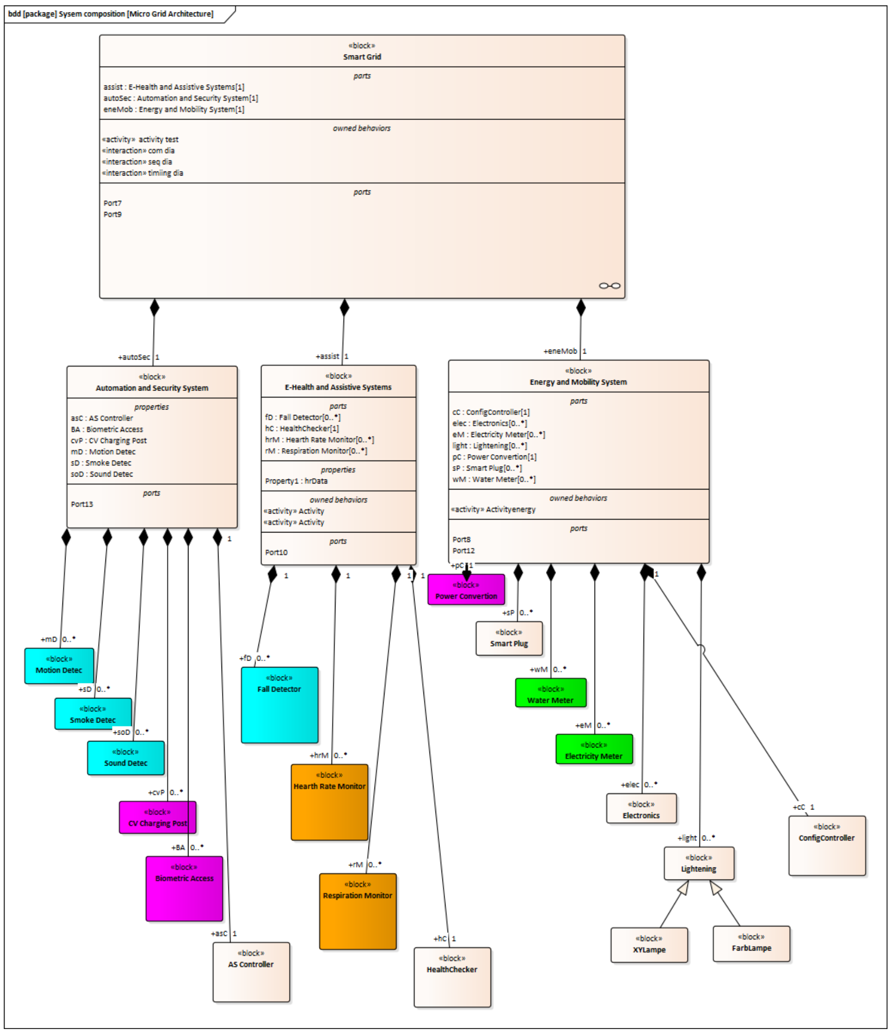


Figure A.3.: *Micro Grid Block Definition Diagram*

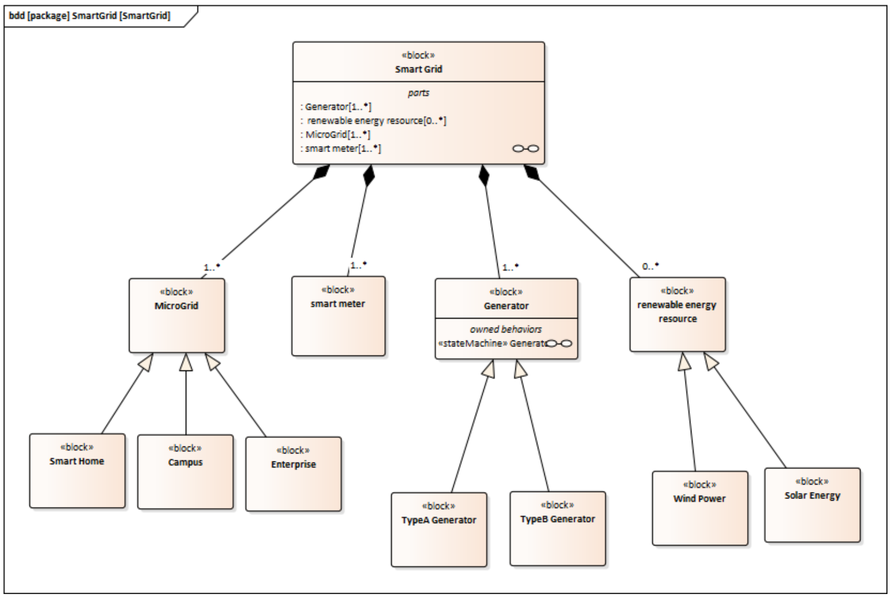


Figure A.4.: *Smart Grid Block Definition Diagram*

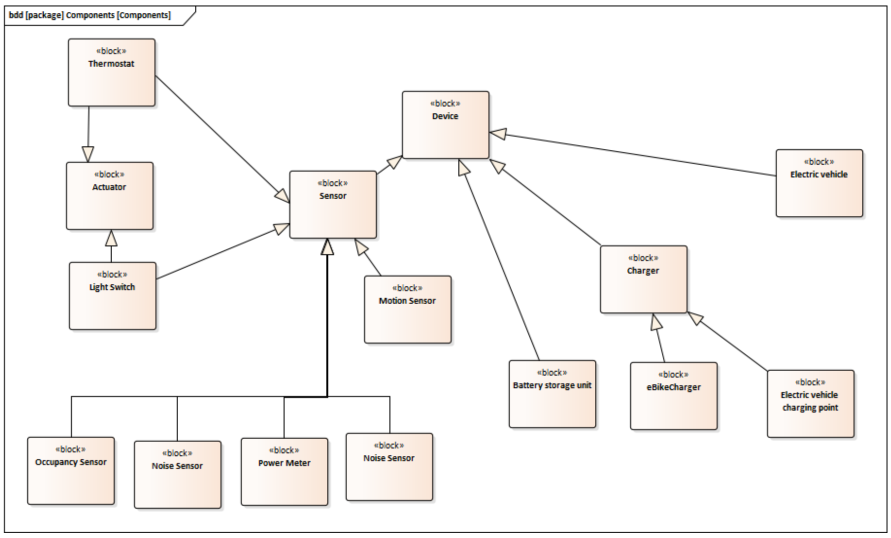


Figure A.5.: *Components Block Definition Diagram*

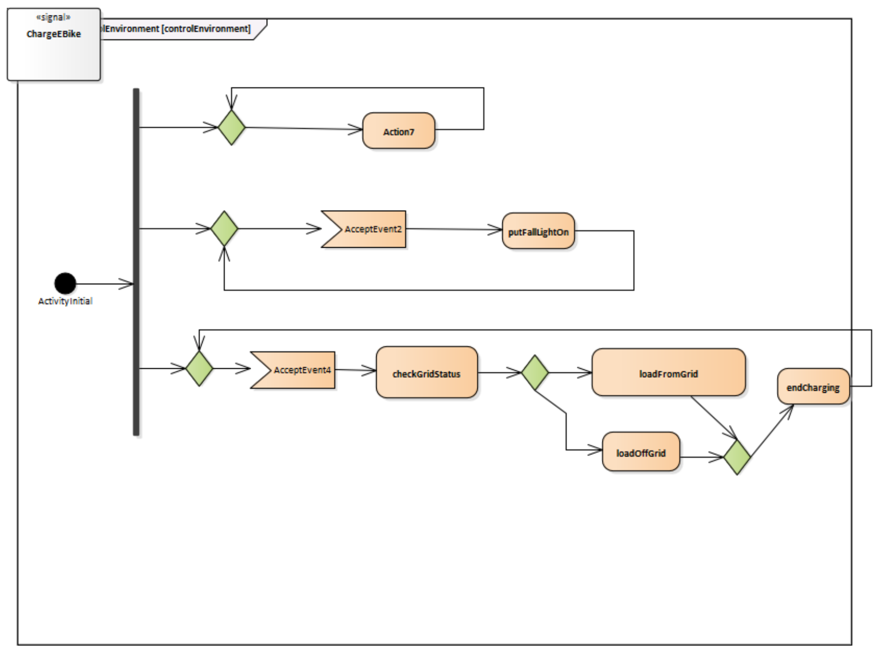


Figure A.6.: *Control Environment One Activity Diagram*

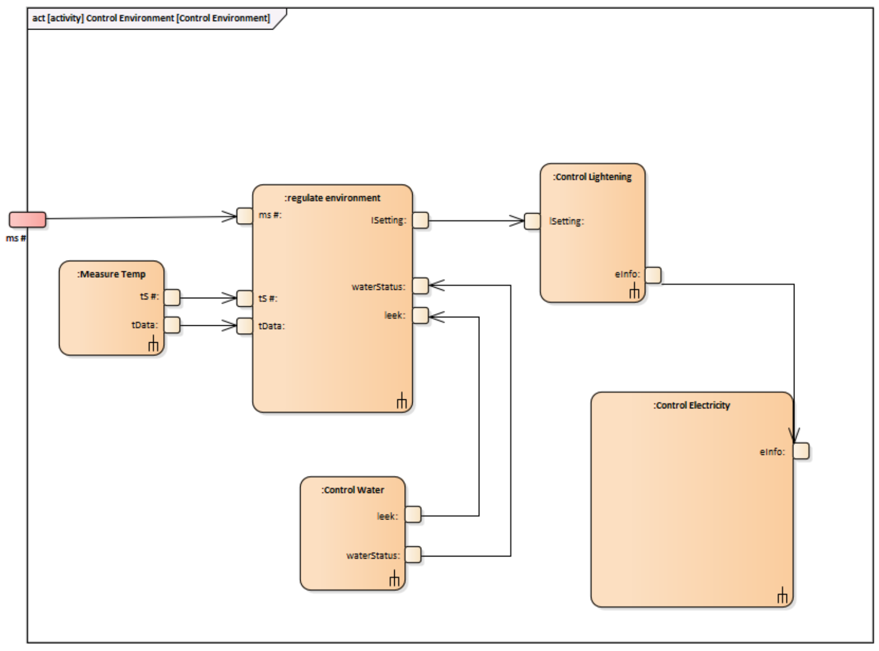


Figure A.7.: *Control Environment TWO Activity Diagram*

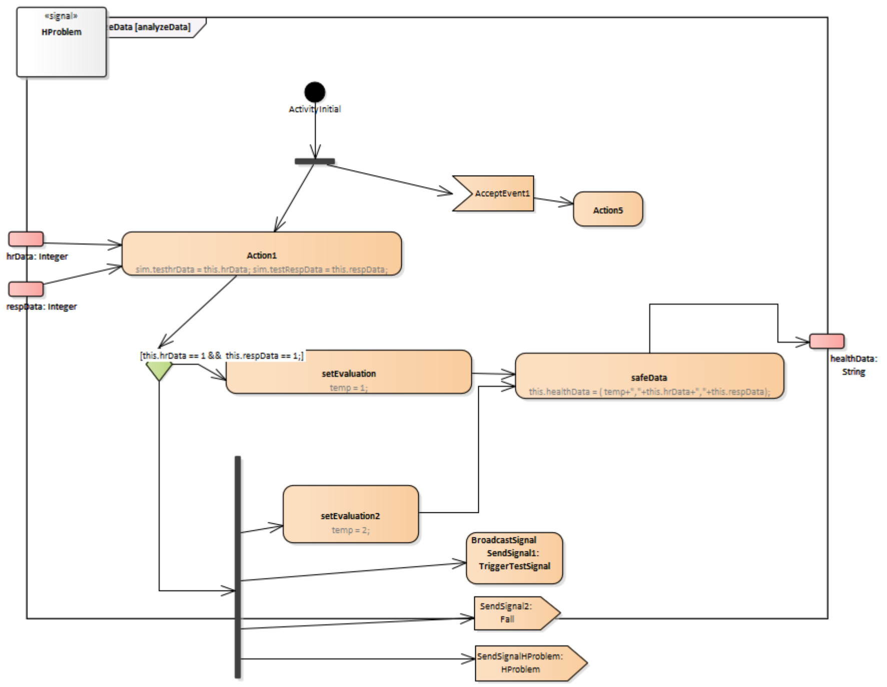

«signal»
HProblem
eData [analyzeData]
ActivityInitial
AcceptEvent1
Action5
hrData: Integer
respData: Integer
Action1
sim.testhrData = this.hrData; sim.testRespData = this.respData;
[this.hrData == 1 && this.respData == 1;]
setEvaluation
temp = 1;
safeData
this.healthData = (temp+","+this.hrData+","+this.respData);
healthData: String
setEvaluation2
temp = 2;
BroadcastSignal SendSignal1: TriggerTestSignal
SendSignal2: Fall
SendSignalHProblem: HProblem

Figure A.8.: *Analyze Data Activity Diagram*

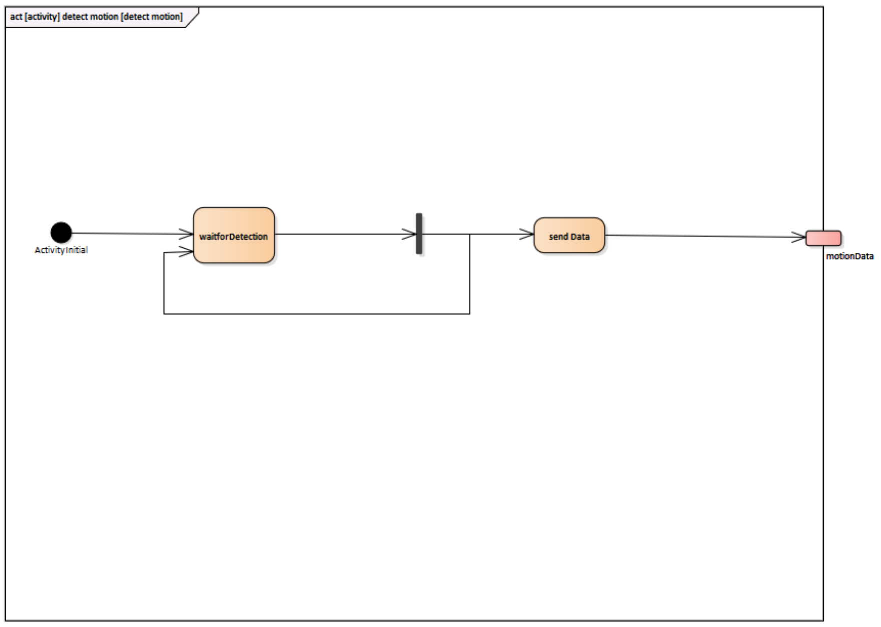


Figure A.9.: *Detect Motion Activity Diagram*

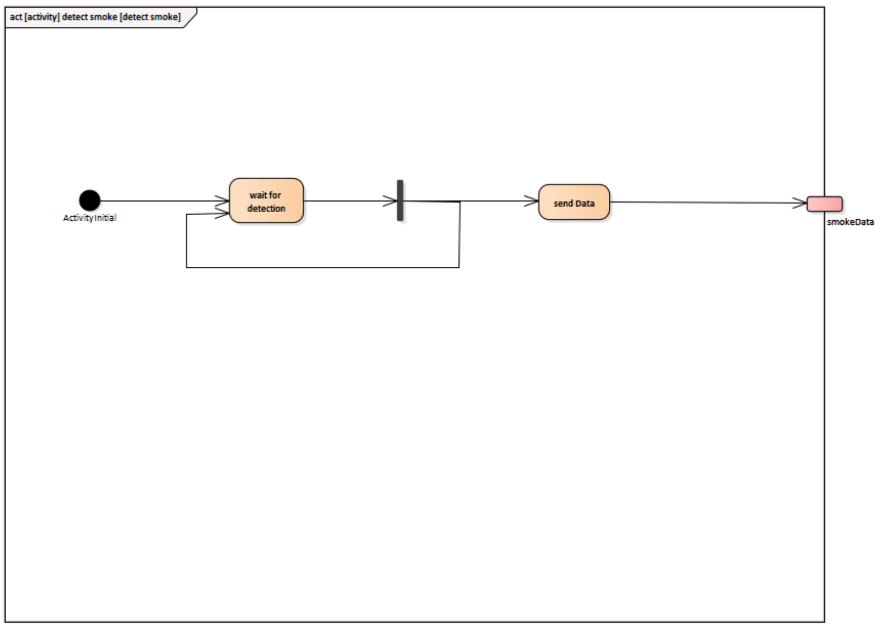


Figure A.10.: *Detect Smoke Activity Diagram*

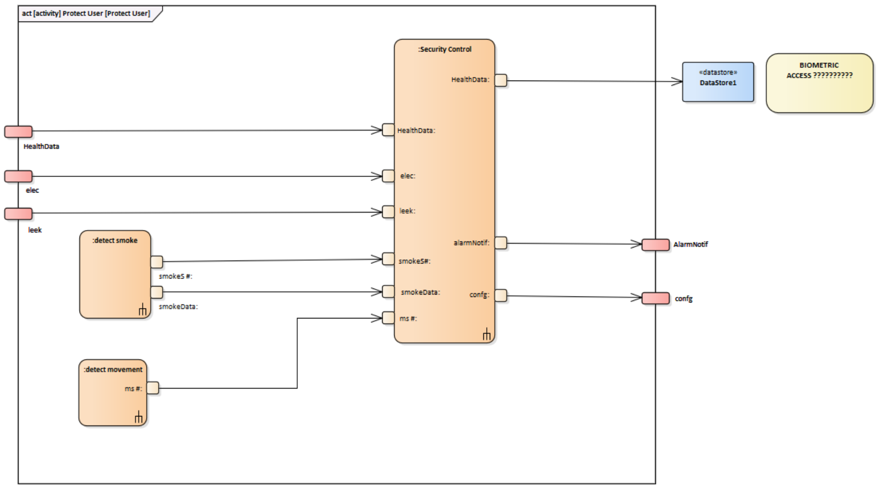


Figure A.11.: *Protect User Activity Diagram*

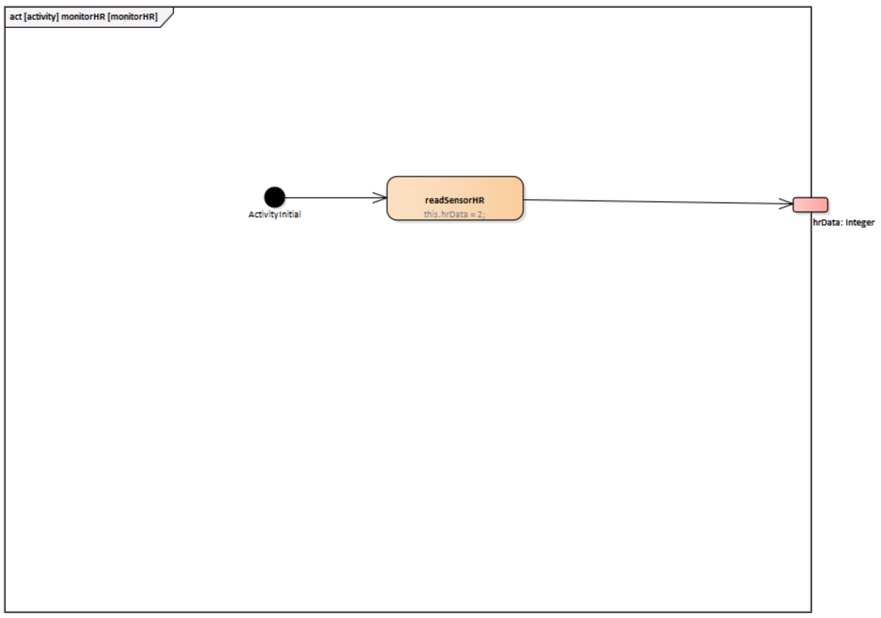


Figure A.12.: *Monitor Heart Rate Activity Diagram*

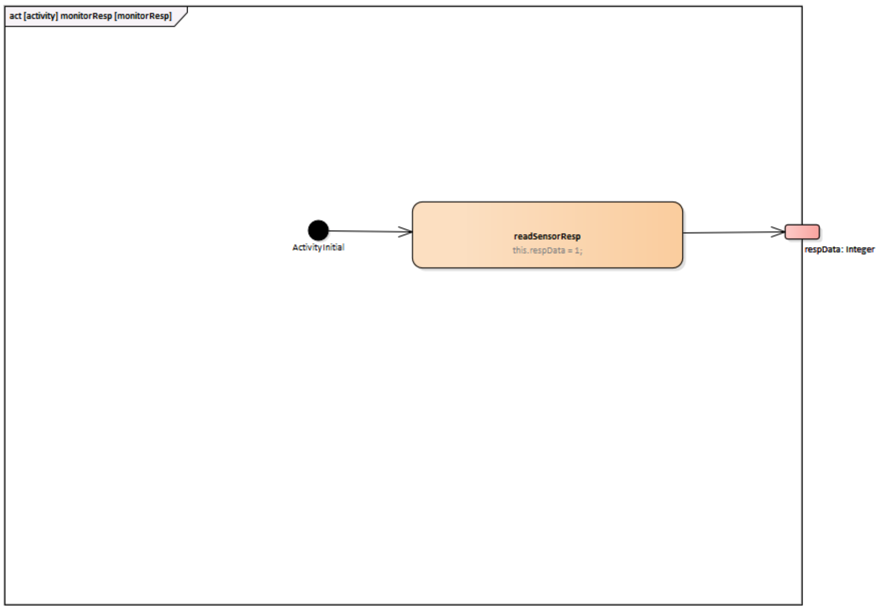


Figure A.13.: *Monitor Respiration Activity Diagram*

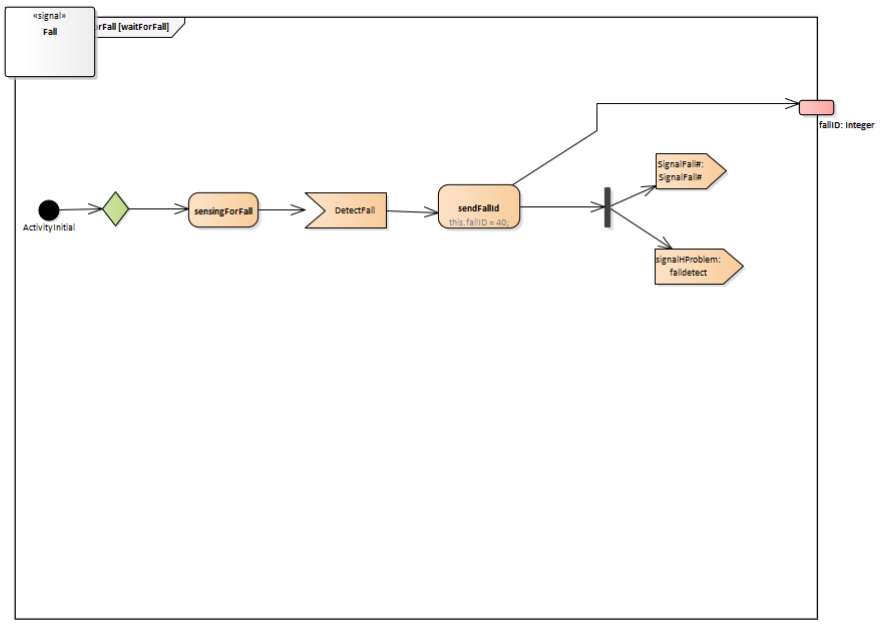


Figure A.14.: *fallsensor Activity Diagram*

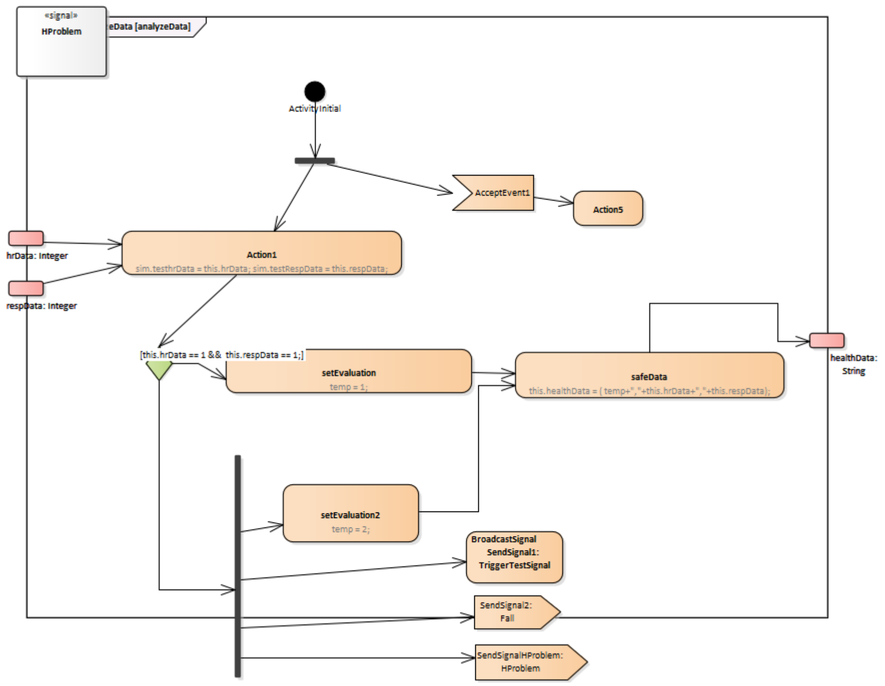

«signal»
HProblem
eData [analyzeData]
ActivityInitial
AcceptEvent1
Action5
hrData: Integer
respData: Integer
Action1
sim.testhrData = this.hrData; sim.testRespData = this.respData;
[this.hrData == 1 && this.respData == 1;]
setEvaluation
temp = 1;
safeData
this.healthData = (temp+","+this.hrData+","+this.respData);
healthData: String
setEvaluation2
temp = 2;
BroadcastSignal SendSignal1: TriggerTestSignal
SendSignal2: Fall
SendSignalHProblem: HProblem

Figure A.15.: *Analyze Data Activity Diagram*

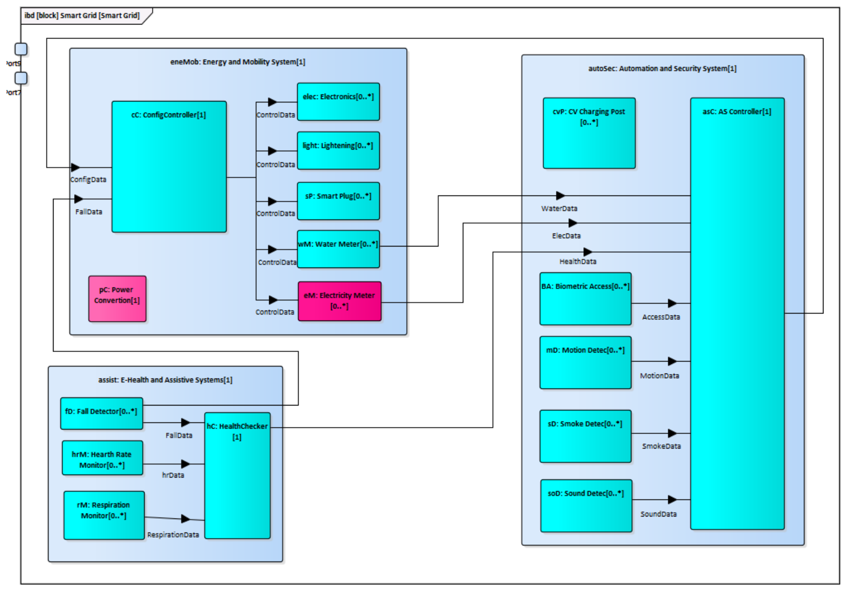


Figure A.16.: *Smart Grid Package Diagram*

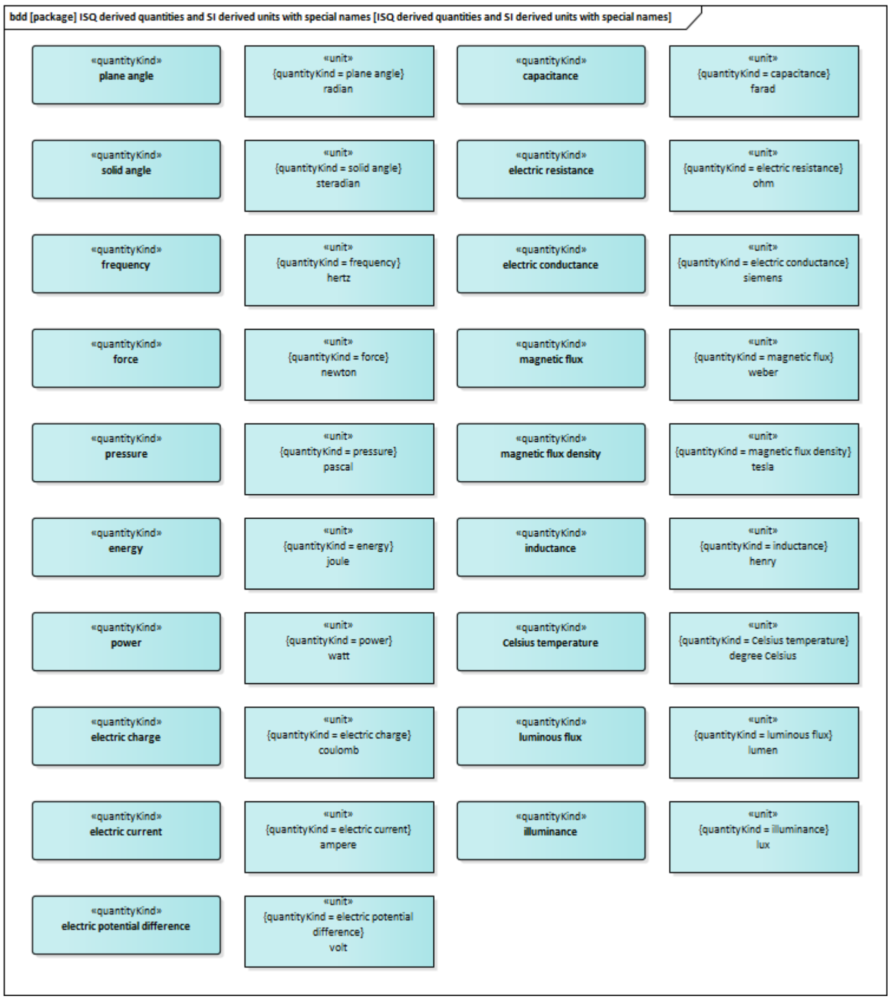

bdd [package] ISQ derived quantities and SI derived units with special names [ISQ derived quantities and SI derived units with special names]
«quantityKind» plane angle
«unit» {quantityKind = plane angle} radian
«quantityKind» capacitance
«unit» {quantityKind = capacitance} farad
«quantityKind» solid angle
«unit» {quantityKind = solid angle} steradian
«quantityKind» electric resistance
«unit» {quantityKind = electric resistance} ohm
«quantityKind» frequency
«unit» {quantityKind = frequency} hertz
«quantityKind» electric conductance
«unit» {quantityKind = electric conductance} siemens
«quantityKind» force
«unit» {quantityKind = force} newton
«quantityKind» magnetic flux
«unit» {quantityKind = magnetic flux} weber
«quantityKind» pressure
«unit» {quantityKind = pressure} pascal
«quantityKind» magnetic flux density
«unit» {quantityKind = magnetic flux density} tesla
«quantityKind» energy
«unit» {quantityKind = energy} joule
«quantityKind» inductance
«unit» {quantityKind = inductance} henry
«quantityKind» power
«unit» {quantityKind = power} watt
«quantityKind» Celsius temperature
«unit» {quantityKind = Celsius temperature} degree Celsius
«quantityKind» electric charge
«unit» {quantityKind = electric charge} coulomb
«quantityKind» luminous flux
«unit» {quantityKind = luminous flux} lumen
«quantityKind» electric current
«unit» {quantityKind = electric current} ampere
«quantityKind» illuminance
«unit» {quantityKind = illuminance} lux
«quantityKind» electric potential difference
«unit» {quantityKind = electric potential difference} volt

Figure A.17.: *ISQ quanities and units Package Diagram*

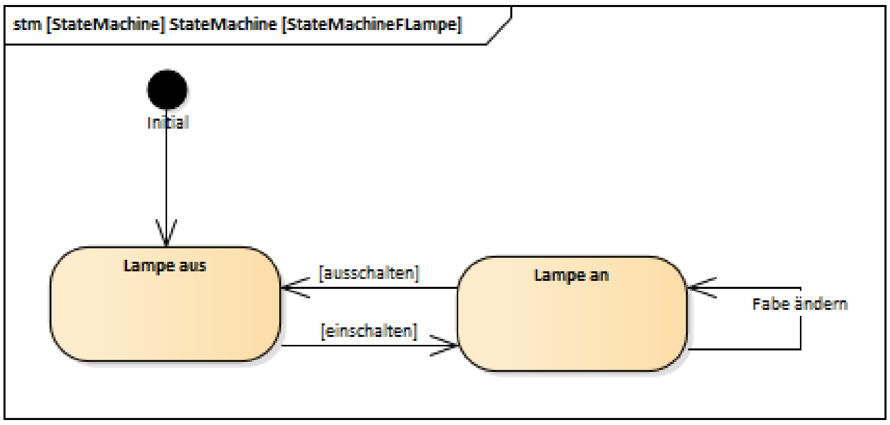


Figure A.18.: *Light State Machine Diagram*

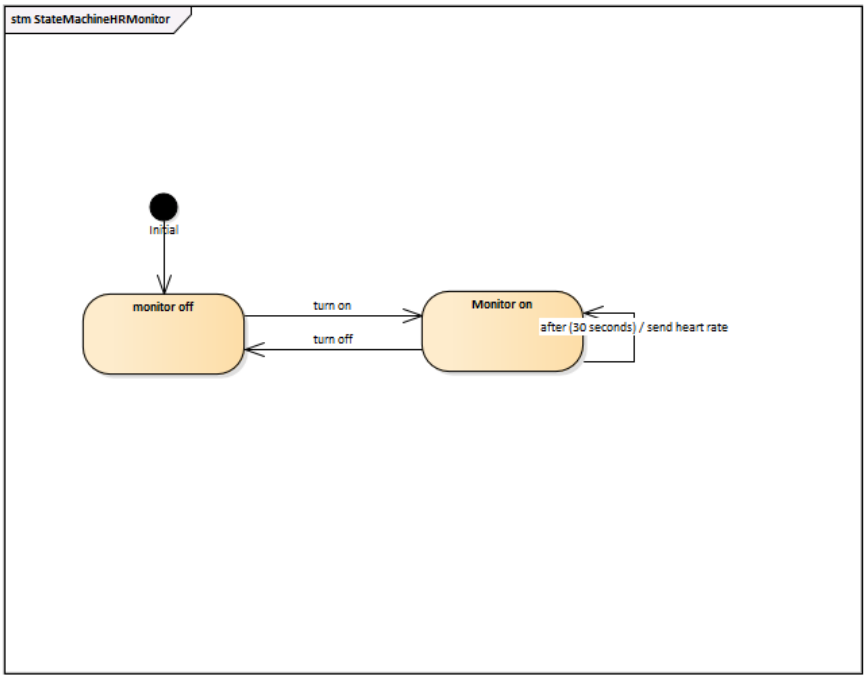

stm StateMachineHRMonitor
Initial
monitor off
turn on
turn off
Monitor on
after (30 seconds) / send heart rate

Figure A.19.: *Heart Rate State Machine Diagram*

Appendix B

Curriculum Vitae

Berufserfahrung

2016 bis heute	**Wissenschaftlicher Mitarbeiter** TU Kaiserslautern Fachbereich Informatik, Arbeitsgruppe Entwicklung Cyber-physikalischer Systeme
2015–2016	**Wissenschaftliche Hilfskraft** TU Kaiserslautern Fachbereich Informatik, Arbeitsgruppe Entwicklung Cyber-physikalischer Systeme
2013–2015	**Wissenschaftliche Hilfskraft** DFKI Innovative Fabriksysteme
2011–2013	**Wissenschaftliche Hilfskraft** DFKI SmartFactory GmbH
2010–2012	**Wissenschaftliche Hilfskraft** TU Kaiserslautern Lehrstuhl Produktionswirtschaft

Akademische Ausbildung

2013–2016 **Master of Science in Angewandte Informatik** TU Kaiserslautern

Anwendungsgebiet:	*Fahrzeugtechnik*
Masterarbeit:	*A Framework For Generation And Co-Simulation Of Domain-Specific Languages From SysML*

2009–2013 **Bachelor of Science in Angewandte Informatik** TU Kaiserslautern

Anwendungsgebiet:	*Produktions- und Fahrzeugtechnik*
Bachelorarbeit:	*Power Profiling for Low-Power Embedded System*

Schulausbildung

2000-2009 **Abitur** Hannah-Arendt-Gymnasium, Hassloch
Leistungskurse: Mathematik, Physik, Erdkunde.