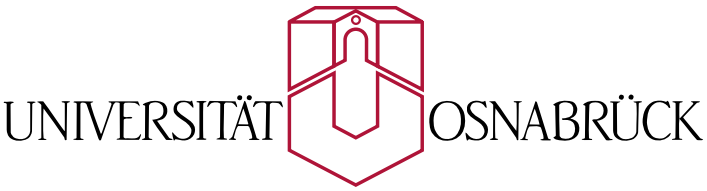


POWER CONSUMPTION MODELING AND ESTIMATION FOR SOFTWARE APPLICATIONS IN MODEL-DRIVEN DEVELOPMENT OF EMBEDDED SYSTEMS

Marco Schaarschmidt

Dissertation zur Erlangung des Doktorgrades (Dr. rer. nat.)
des Fachbereichs Mathematik/Informatik
der Universität Osnabrück

Eingereicht am: 27.07.2023
Disputation am: 06.10.2023

Erste Gutachterin:	Prof. Dr.-Ing. Elke Pulvermüller
Zweiter Gutachter:	Prof. Dr.-Ing. Clemens Westerkamp
Dritter Gutachter:	Assoc. Prof. Dr. Hany Elgala

Acknowledgments

Writing a thesis is surely an intense experience. Over the past years, I have had the honor and joy of working with many inspiring, open-minded, and wise people who have always motivated me to keep going and stick to my research ideas. I am very grateful for the help of every one of them.

First and foremost, I would like to express my gratitude to *Prof. Dr.-Ing. Elke Pulvermüller* for supervising this thesis, the trust I received from her, and our discussions over the past few years. I am also grateful for all the scientific and methodical feedback on my work. I also want to thank *Prof. Dr.-Ing. Clemens Westerkamp* for co-advising this thesis and the opportunity to work on many exciting research projects, which helped me stay open-minded as a researcher while maintaining a broad perspective. Next, I want to thank *Assoc. Prof. Dr. Hany Elgala* for accepting the role of the third reviewer.

I extend my deepest gratitude to *Prof. Dr.-Ing. Michael Uelschen* for his trust and support in writing this thesis and the work on scientific publications. It was fun - well, mostly. You never stopped motivating and pushing me toward the end of this thesis. I am also grateful for the valuable discussions and brainstorming sessions about research ideas and problem statements. To use the wording of a computer scientist: I often experienced our conversations as the needed break command to exit the 77 thought loops that have emerged. I truly hope we can continue our research and tackle the following 50 ideas left on our list.

As part of the Software Engineering Lab, I enjoyed working with my colleagues on challenging research projects. I thank *Alexander Grunwald* and *Timo Thurow* for fruitful discussions about physics, electrical engineering, and embedded systems. Next, I would like to thank *Jannis Budde* and *Simon Balzer*, who refined some concepts and provided proof-of-work implementations with their Master's and Bachelor's theses.

Additionally, I want to thank *Lars Huning* and *Michael Spieker* for many fruitful discussions about ideas and concepts related to model-driven development.

Finally, I want to thank my parents, my brother *Robin*, *Stephi*, and *Isabella*, for their endless faith, understanding, motivation, and support through all these years. There have been many ups and downs along the way to this thesis, and I'm fully aware that my mood hasn't always been the best. But let's be clear: This work would not have been possible without you!

A big **thank you** to each and every one of you!

Abstract

Nowadays, embedded systems are ubiquitous and inherent in almost all areas of life. In recent years, trends like the *Internet of Things (IoT)* have been a primary driver for the growing embedded systems market. Many of those IoT devices are battery-powered and more resource-constrained. In addition to economic constraints like total costs and short time-to-market, technical constraints lead to multiple challenges in embedded software development. For battery-powered systems, electrical energy is one of the most critical constraints. For instance, uncontrolled power consumption or an exhausted energy source caused by software applications may lead to failure and costly damage to the device or the environment. Developers often lack knowledge and suitable design concepts to specify, implement, and evaluate energy-efficient software applications. Additionally, constantly changing technologies, extensive functionalities, and various requirements further increase the complexity of embedded software applications while making their development a critical and complex task. To manage the complexity of software applications and the development process, methodologies such as *Model-driven Development (MDD)* have gained importance. However, power-related non-functional aspects are insufficiently considered in MDD.

This thesis addresses the aforementioned gaps and presents a novel framework for energy-aware software design patterns. Developers and engineers may use the framework to specify and describe design patterns addressing power-related issues of software applications. The introduced design pattern template provides a set of metrics to describe possible energy savings and the effort-saving ratio when applying a design pattern. The template also contains a unified graphical representation to visualize the effects of design patterns. In addition, a first catalog of energy-aware design patterns is provided, which may be used to design software applications in MDD and traditional development.

To further enhance the development of energy-efficient software applications in MDD, this thesis also introduces a novel power consumption estimation approach for models based on the *Unified Modeling Language (UML)*. The approach is specifically designed for early development stages when optimizations are most effective. A concept for hardware component models is presented, which can be integrated into the software application model. With the provided UML profile, aspects related to power and timing can be modeled. In addition, methods for indirect and direct power analysis are introduced. While the indirect power analysis is based on simulated hardware behavior, the direct power analysis relies on a real hardware platform and a measuring device. Along with the novel and formal description of energy bugs, software applications can be evaluated and energy-related issues detected.

A real-world example of an IoT sensor node and a proof-of-concept implementation of the power consumption estimation approach illustrate the application of proposed modeling and estimation concepts. Moreover, the detection of energy bugs is demonstrated, and the accuracy of the analysis methods is compared. Additionally, the overall performance of the direct power analysis is investigated in depth. The results have shown that the concepts and approaches are suitable for analyzing and predicting the power consumption of software applications in early development phases. Furthermore, the process can be integrated into existing development workflows to support developers using MDD to design energy-efficient software applications.

Contents

Chapter 1

Introduction

This chapter introduces the field of embedded systems and motivates the need for energy-aware software applications. It also discusses how model-based approaches can overcome the challenges of developing energy-aware software applications and explains the importance of considering energy and power-related aspects in the early evaluation. The resulting problem statement is framed by the elaboration of challenges from which this thesis's scope of research and research questions are derived. Furthermore, limitations are discussed, and the structure of this thesis is outlined.

1.1 Motivation

The global market for embedded systems has grown steadily over the past few decades due to the number of different applications. Nowadays, embedded systems are ubiquitous and can be found in domains such as automotive, healthcare, industrial automation, telecommunication, the *Internet of Things (IoT)*, and the *Industrial Internet of Things (IIoT)*. In particular, IoT and IIoT, where embedded systems are used for smart homes, smart cities, agriculture, smart factories, and environmental monitoring, have the highest expected growth [2, 105, 423]. Researchers assume a continuing expansion of IoT devices over the following years. A study proposed in [400] expects an increase from 8 billion devices in 2020 to more than 25 billion by 2030. The authors of [80, 401] forecast IoT devices to represent 50 % of all networked devices in the year 2023 using short-range technologies, such as Wi-Fi, Bluetooth, and Zigbee. In a study released by Transforma Insights [257], an increase from 13.2 billion IoT devices in 2022 to 34.4 billion in 2032 is expected, corresponding to an annual growth rate of 10 %. Friedli et al. (2016) [126] estimate the number of mains-powered IoT devices in categories such as smart appliances, roads, lighting, and home assistants to reach 5.7 billion by 2025, with an expected standby power consumption of 46 TWh. The number of battery-powered IoT devices is expected to reach 23 billion units in the same year [395]. In addition, the trend toward more electric-powered devices across all sectors and the change to alternative energy sources raise the risk of power outages. This affects the overall availability of electrical power, making it an increasingly important resource. As energy prices continue to rise [416], the power consumption of these devices becomes a critical design constraint.

Embedded systems are generally resource-constrained, especially for domains relying on low-power systems such as IoT. In addition to general economic-related constraints like total costs and short time-to-market, functional safety and technical constraints such as the

processing power, memory size, or battery capacity lead to multiple challenges in software engineering. Supplementary to the mentioned constraints and the increased functionality of modern software applications, software developers and engineers have to deal with the growing complexity of embedded system designs at a low level due to the variety of processor architectures, communication interfaces, and the growing number of proprietary hardware with distinct functionalities. Especially for battery-powered IoT devices, energy and power-related *Non-functional Requirements (NFRs)* for hardware and software need to be defined at the beginning of the development since the battery capacity is one of the most critical factors for the operational lifetime of the device. The supply of power can be a major challenge if those devices are placed in harsh environments or buried underground [146, 409]. From an economic and technical perspective, replacing or recharging the power source for these devices is often either impossible, impractical, or results in higher costs.

Although energy consumption is an invisible property, it can be a significant bottleneck of embedded systems [35]. Up to 80 % of the total energy consumption is generated by the software application [231] since it drives and controls hardware components and, thus, directly affects the energy-related behavior of the systems. In the past, energy and power consumption aspects were addressed primarily at the hardware level, resulting in more energy-efficient hardware components. So far, little attention has been given to the software layer that controls and directs most hardware activities. In fact, as a property of hardware components, the static energy demand can be addressed by improved hardware designs. In contrast, optimizing the software application can address the dynamic energy demand during runtime. Energy awareness is often completely ignored in the software development process [123] because developers and engineers may be unaware of the causes of high energy consumption and lack knowledge of how to reduce the energy impact of software applications [38, 289, 301]. Since the impact of the software application on energy consumption is often unknown, it is essential to consider power-related NFRs in early design phases, where changes are more effective [375]. However, even if the definition of NFRs related to energy and power in early development phases prompts developers and engineers to design energy-efficient software applications, the traditional software development process is not designed to address such NFRs at an early development stage. The software development process typically focuses on functional aspects, while non-functional aspects are tested and evaluated later in the development cycle. At this point, the software application and hardware platform are close to their final states, and the software application can be executed on the embedded system. Addressing non-functional issues such as power consumption, energy efficiency, and real-time in later development phases may lead to extensive re-design and re-implementation phases if changes at the software architecture level or requirements level are needed [33]. These processes may extend the development time and lead to higher costs, schedule delays, lost productivity, damaged customer relations, or missed market windows [361]. In addition, no approaches or tools exist for estimating the power consumption and detecting potential power-related misbehavior of software applications in early development phases where the target hardware platform may not be available or defined.

Indeed, model-driven approaches offer the opportunity to address and improve software quality in early development phases [292]. As previous paradigm shifts in embedded software engineering have led to higher levels of abstraction, approaches such as *Model-driven Development (MDD)* for designing and developing software applications have received more attention. MDD uses models and modeling techniques for the software development process to not only achieve higher levels of abstraction but also to manage the complexity of software applications,

e.g., for embedded systems. Furthermore, MDD also increases the formalization of software development activities and tasks so that processes can be automated [152, 369], which also raises productivity [193]. Such automation allows developers and engineers to focus on the application logic, behavior, and program flow of software applications and helps overcome the challenges of the development process [56, 326]. Due to the formalization, models can be exchanged between domains and reused with concepts like model-to-model and model-to-text transformation, which increases developer productivity even further. Additionally, MDD enables an evaluation of requirements at the architecture level in early stages of the development process, e.g., using *Model-based Testing (MBT)* techniques. Exemplary for software applications used in airplanes, [153] compared an architecture-modeling approach with existing development paradigms and found that 70 % of the software defects are located at the requirement or design level. While more than 50 % were identified during the hardware/software integration as part of later development phases, less than 10 % of these defects were detected in their respective phases. Due to this, the rework costs were 100 times higher than the costs for correcting the errors at the levels they occurred.

McKinsey [92] confirms the increasing complexity of software development within embedded systems and suggests the consequent application of *Domain-driven Design (DDD)* and a ubiquitous language. As a ubiquitous language, the *Unified Modeling Language (UML)* [275] may be used by both DDD and MDD [114]. As the most used modeling language in the embedded software industry [7], UML provides concepts for object-oriented modeling of structural and behavioral aspects of software applications and provides graphic notation techniques so that software developers can focus on the application design, behavior, deployment, and program flow. Additionally, with UML profiles, UML provides a generic extension mechanism to include domain-specific concepts in the definition of models using stereotypes, tag definitions, and constraints. Existing notations and specifications may be considered to introduce energy awareness in UML. The *Modeling and Analysis of Real-Time and Embedded systems (MARTE)* UML profile [278] extends the UML and provides a set of modeling concepts, data types, and notations to support the modeling and analysis of real-time and embedded systems. MARTE also enables the modeling of *Non-functional Properties (NFPs)*, such as timing constraints, performance, and schedulability, as well as basic power consumption and dissipation aspects as static values.

It is crucial to consider both static and dynamic power and energy consumption aspects for an energy-efficient system. Since the software application is responsible for hardware activities during runtime, the advantage of addressing the software application to reduce the (dynamic) energy consumption of an embedded system is obvious. In conjunction with the fact that changes and optimizations of the software application are considered to be more effective at higher levels of abstraction [375], design flaws and behavioral issues should be identified early in development to minimize costs and development time. This is especially true for energy- and power-related issues, which, as noted earlier, are typically discovered in later phases of the development process. Although many efforts in the research areas of software engineering and software development address energy-efficient software applications and energy-aware development, there are still a number of unresolved high-potential topics. In software engineering, for instance, such topics include the exploration of suitable design concepts, models, patterns, and strategies to address energy- and power-related aspects [38]. MDD, however, lacks concepts, methods, and tool support for estimating power-related non-functional aspects in early design and development phases. These unresolved topics will be addressed within this thesis and further elaborated in the next section.

1.2 Problem Statement and Scope of Research

As motivated in the previous Section 1.1, developing software applications for embedded systems is a challenging task. In addition, developers and engineers have to deal with the increasing complexity of embedded software applications caused by highly dynamic and constantly changing technologies, extensive functionalities, protocols, and requirements [49, 282, 292, 369]. Especially for low-power and battery-powered embedded systems, energy and power efficiency are the most critical quality attributes determining the operational lifetime of the device, which has to be addressed by engineers and developers. This also implies that software applications play a central and critical role. For instance, the system will fail if the energy source is exhausted too quickly due to a non-optimized software application or uncontrolled energy consumption, which may result in costly damage, e.g., a loss of the device or harm to the environment [202]. Regarding power consumption, *Steve Furber*, the designer of the original ARM microprocessor, stated in an interview in 2010 [60]:

> *"If you want an ultimate low-power system, then you have to worry about energy usage at every level in the system design, and you have to get it right from top to bottom, because any level at which you get it wrong is going to lose you perhaps an order of magnitude in terms of power efficiency. The hardware technology has a first-order impact on the power efficiency of the system, but you've also got to have software at the top that avoids waste wherever it can. [...] Do programmers really have any understanding of how much energy their algorithms consume?"*

This statement underlines the importance of optimization and evaluation at higher levels of abstraction, e.g., the software architecture level, where changes are expected to be most effective [143, 375]. This is also confirmed by researchers who argue that energy consumption needs careful consideration [35] and should be studied across all software development phases [123]. Understanding the impact of the software application on properties such as power consumption and execution time requires developers to have in-depth information on behavioral aspects of the software application, in particular hardware-software interactions. The availability of energy-efficient hardware components indicates that power consumption is well-addressed by developers and researchers at the hardware level. However, little attention has been given to the software level that controls and directs most hardware activities, and the energy-related impact of software applications is often unknown. In [38], the authors stated that *"[m]ost architects and developers are unaware of energy efficiency as a quality attribute of concern, and hence do not know how to go about engineering and coding for it. More fundamentally, they lack an understanding of energy efficiency requirements"*. This statement highlights important research gaps in energy-aware software engineering. For instance, new concepts have to be developed to enhance the understanding of energy efficiency requirements, how they are derived, and how they relate to software and system behavior. Developing energy-efficient software encompasses many aspects of software development. This includes the optimized use of compilers and (object-oriented) programming languages and the optimization of the software architecture and behavior, e.g., the interaction between software modules and hardware components. In software engineering, design patterns, as proven best practices, might be used to overcome the complexity of applications and address energy-efficiency aspects. However, there is an ongoing lack of suitable design concepts such as tactics or patterns [38]. Additionally, and to the best of our knowledge, only limited work towards software design

patterns exists, which directly describes the effect on energy efficiency and power consumption when applied to the design of software applications.

With MDD as the main methodology used in this thesis, the software development process is mainly driven by the use of models. Consequently, the consideration of power-related aspects should be part of the MDD process and based on the defined models. Quantitative results obtained by evaluations during the MDD process may be used to adapt the software architecture and design to fulfill energy- and power-related NFRs already in early development phases. Since the impact of software applications depends highly on the hardware platform, an analysis of NFRs, especially for power- and energy-related aspects, is typically carried out by engineers and developers in later development cycles where the software application and the hardware platform are more advanced and functional testing has been completed [235, 260, 367]. To still be able to perform early evaluations of, e.g., power constraints, a suitable energy model has to be developed [35]. Additionally, to execute performance evaluations w.r.t. energy and power consumption aspects, concepts to simulate and analyze the behavior of software application models must be supported by MDD tools and integrated into the development workflow of developers and engineers. Commercially available MDD tools such as MathWorks MATLAB [380], MathWorks Simulink [383], and IBM Engineering Systems Design Rhapsody – Developer[1] [164] provide simulation environments and support the execution of model-based test cases. Additionally, concepts and extensions for MathWorks MATLAB [382] and IBM Rhapsody [165] exist, providing techniques to execute test cases on, e.g., embedded systems. However, the evaluation provided by the aforementioned MDD tools is focused on functional aspects of software applications and lacks support to analyze and evaluate NFRs. Even while not directly mentioning MDD, *Steve Furber* also addresses the lack of tool support during his interview [60]:

> *"[...] programmers will not be able to afford to be ignorant about the energy cost of the programs they write. [...] You need tools that give you feedback and tell you how good your decisions are. Currently the tools don't give you that kind of feedback."*

Currently, the evaluation of embedded software applications in MDD requires the integration of platform-specific source code. Moreover, it is based on repetitive steps, including manual effort such as editing, compiling, and flashing the auto-generated source code for one or multiple targets if, for instance, the most suitable system components, e.g., as sensors or actuators, should be identified. This process is also denoted as *edit-cross-compile-flash-debug* cycle [30, 393].

Besides the lack of tool support, the direct execution and evaluation of software applications on target hardware platforms is part of later development phases when prototypes are available. In early development phases, however, frequent hardware component changes still occur. The effort necessary to evaluate non-functional aspects in this manner may result in a bottleneck of the development process and may cause additional time delays and costs. This can be compensated by a faster evaluation process in early phases, which may include, e.g., rapid prototyping principles. It is essential to have a proper set of methods and tools that help software developers in early design phases. These tools should enable the evaluation of software applications regarding power-related NFRs by estimating the power consumption and detecting possible power-related misbehavior. To the best of our knowledge, in MDD, no approach exists for an early and straightforward power consumption estimation of software

[1] To improve readability, the abbreviation *IBM Rhapsody* is used in the following chapters of this thesis.

applications. Additionally, since embedded systems in domains such as IoT typically consist of multiple sensors, actuators, and at least one communication interface, the power consumption estimation approach should encompass the entire system.

With the right concepts, methods, and tools, developers and engineers are fully aware of the software application's system-wide energy footprint. They may be able to design and evaluate energy-efficient software applications in an enhanced manner. Accordingly, the following thesis statement emerges from considering open research subjects and the aim of this thesis.

> ***Thesis Statement:*** *"Developers and engineers are able to design energy-efficient software applications and evaluate energy-related requirements in early development phases of MDD if appropriate best practices exist and powerful analysis methods with tool support are provided."*

The challenges resulting from the problem statement are addressed in the following Section 1.2.1. The research questions derived from the identified challenges are presented along with the scientific contributions of the thesis in Section 1.2.2.

1.2.1 Research Challenges

Modeling and estimating a software application's impact on the overall system's power consumption is challenging, especially in early development phases, where the prototype hardware platform may not be available or defined. Different challenges exist along the way to achieve a power consumption estimation workflow for software applications in MDD, which can be grouped into four broad categories:

Challenge 1: Definition of Power- and Energy-related Requirements
The description and differentiation of correct and faulty power- and energy-related behavior is crucial for testing and evaluating the NFPs of embedded systems and, simultaneously, a complex and challenging task due to the unique characteristics of these systems. Effective testing and evaluation of power- and energy-related behavior requires a specific and more formal definition of NFRs to determine whether the intended behavior meets the expectations. In [34, 293, 295], a broad classification and definition of power-related misbehavior for certain domains and device classes, e.g., smartphones, are provided. For example, Banerjee et al. (2014) [34] mentioned that power-related misbehavior denoted as *suboptimal resource binding* is defined as *"binding resources too early or releasing them too late causes them to be in [a] high-power state longer than required"*. These and similar statements are primarily formulated in a natural language, while a formal definition of such energy and power-related misbehavior is missing. For the rest of this thesis, energy and power-related misbehavior are referred to as *energy bugs*. A formal definition of energy and power-related NFRs would have a strong interrelationship with energy bugs, where one may be derived from the other. In addition, such a formal definition may significantly contribute to automatic analysis. It is not known how to formally define energy and power-related NFRs and describe energy bugs so that they can be interpreted and processed by both humans and machines.

Challenge 2: Best Practices of Energy-aware Design
When developers and engineers address power-related issues during software application design and implementation phases, the availability of documented best practices is essential to implement the most appropriate solution. Such paradigms are typically referred to as *software*

design patterns. By using structural, creational, and functional design patterns, developers and engineers can optimize and improve the software application in early phases while avoiding time delays and increasing costs due to additional re-design and optimization cycles. However, existing design pattern templates do not provide categories or fields to address non-functional aspects such as power and energy consumption. To be able to calculate and estimate the effects of a software design pattern, a design pattern template is needed to describe such energy-aware software design patterns uniformly. With a focus on non-functional aspects, such a uniform description allows the developer to compare the effects of different design patterns without having to implement and test each variation and select the most suitable design pattern with the highest positive effect.

Challenge 3: Design and Modeling of Power-related Properties
To model power-related aspects of the software application in MDD, properties of hardware components and their dynamic behavior must be considered. Approaches found in the literature and discussed in Section 2.8 (p. 65 ff.) lack the capabilities to combine those power-related aspects with the software application model. However, while focusing on the software perspective, the integration is necessary to consider the software application's dynamic behavior during simulation or execution and hardware-software interactions for the analysis process. So far, it is unknown how software application models can be extended and which techniques are suitable to model power-related aspects in MDD. The additional effort to model power-related aspects should not force software developers to lose their focus on the software design and their actual workflow of the software modeling process. Furthermore, power-related model components must be integrated into the software model without the need to adapt existing structures or restrict intended functionalities. They also have to be removed before the final source code is generated. Therefore, the challenge is to define a lightweight extension for software application models which can be used to estimate power consumption.

Challenge 4: Simulation and Evaluation of Software Application Models with integrated Energy Models
For the evaluation in early development phases, the simulation of software application models is a significant aspect. By performing simulations as early as possible during development, developers may identify problems that would otherwise remain undetected and become visible in later phases, e.g., field tests. While some tools in the MDD domain offer a simulation environment for evaluation, they focus on functional aspects and do not support an evaluation of non-functional aspects. Furthermore, the hardware behavior must also be part of the simulation environment to reflect hardware-software interactions. So far, no method exists to include power- and energy-related non-functional aspects in the simulation or execution of software application models. Besides the simulation, the analysis of NFRs is also not addressed by current MDD tools. Instead, the analysis is carried out and performed in later development stages, where the time and effort needed to address the misbehavior is significantly higher. However, it is unclear how the analysis may be performed during early modeling. Besides defining concepts for the evaluation process and verification mechanisms, an analysis tool has to be developed which can be integrated into the MDD process. For a proper analysis, an optional solution would be aware of all hardware models and hardware accesses initiated by the software application. Due to the lack of support from MDD tools, the evaluation process must be executed separately, e.g., based on simulation logs. A more promising but

non-trivial approach would be to interconnect the simulation environment with the analysis tool to realize runtime monitoring, which also requires proper instrumentation for a power consumption estimation.

1.2.2 Research Questions and Contributions

To address the challenges introduced in Section 1.2.1, the following *Research Questions (RQs)* have been derived, focusing on the engineering and evaluation of software applications in the early development phases of MDD.

RQ1 – Formal Definition of Energy-related Behavior and Defects: *How should non-functional requirements for energy-related behavior be described?*

Defining a clear and comprehensive set of NFRs is essential when designing energy-aware embedded systems. During the design phase, it is important to precisely describe the expected energy-related behavior so that developers are able to make comparisons during testing and automate the evaluation process. RQ1 is related to *Challenge 1*.

The following contributions to address RQ1 provide fundamental concepts as a basis for further approaches presented in this thesis. A novel formal description of NFRs is presented to accurately specify the expected energy-related behavior of an embedded system. Additionally, a formal specification of *energy bugs* as the description of power-related misbehavior w.r.t. embedded systems is provided. While the specification of energy-related NFRs and the description of energy bugs as violations of such NFRs share the same underlying metrics, power-related misbehavior may be detected and classified during testing. Furthermore, a precise specification of the environment and system properties by a set of conditions and constraints has been elaborated to describe the range of validity of both NFRs and energy bugs when executing test cases. The contributions are published in [341] and may be summarized as follows:

- The introduction of the two metrics, energy quota E_{qu} and the maximum current demand I_{dmax}, for the formal description of power- and energy-related NFRs and for the specification of boundaries for an energy bug-free system or subsystem. Due to the formal definition of energy bugs, they can be expressed using the same metrics. This enables developers to perform a power analysis and evaluation of NFRs and facilitates the application of automated processes.

- A novel and more comprehensive classification of energy bugs independent of specific device types. The presented classification categorizes energy bugs based on their characteristics and origin, e.g., hardware and software layers.

- The concept of scenarios as a set of conditions and constraints to specify aspects of the environment and the system for the execution of test cases that apply for a specific amount of time. As a modeling approach of the environment and the context for testing, the concept of scenarios also addresses RQ4.

RQ2 – Best Practices and Design of Energy-aware Software Applications: *What are the best practices for energy-aware software applications, and how can they be uniformly described and formalized as design patterns?*

This RQ is related to *Challenge 2* and addresses the problem of how aspects related to power and energy consumption as NFPs of a system may be considered when designing software applications. Furthermore, RQ2 aims to investigate whether design patterns for software applications focusing on power and energy consumption can be described uniformly so that software developers can identify and apply best-practice solutions for their problems.

By applying energy-aware software design patterns, it is expected that the misbehavior of software applications may already be addressed at an early stage of development. The idea to address RQ2 is to provide software developers with a catalog of behavioral design patterns for software applications as reusable paradigms at the design and architecture level, focusing on power-related aspects. For this, a novel framework has been defined to identify and describe energy-aware software design patterns. The contributions are published in [337, 338, 392] and may be summarized as follows:

- The development of a novel framework and a design pattern template for the identification and uniform description of software design patterns. The design pattern template introduces a new section to address the impact and side effects on NFRs, the two metrics energy balance EB_P and efficiency factor η_P, and a uniform graphical representation to outline the power-related and computational behavior of design patterns.

- A catalog of energy-aware design patterns to demonstrate and prove the applicability and expressiveness of the framework. The catalog contains novel and existing design patterns from different areas of software and hardware development.

RQ3 – Joint Modeling of Functional Software Application Models and Energy Behavior: *How can software application models be extended with energy-related hardware characteristics to make the software-related impact visible and traceable?*

RQ3 addresses *Challenge 3* and aims to discover concepts and mechanisms for enhancing software application models to make energy-related features visible during analysis. Additionally, RQ3 refers to the necessary level of detail and characteristics when modeling hardware components and how they may be seamlessly integrated into the software application model.

To model power-related aspects of the system, a description of hardware components and their power-related behavior is required. For this, a specification of *hardware component models* is introduced, which also incorporates an *energy model* to specify the temporal power-related behavior of a hardware component model. For the description of power-related aspects, the novel *Power Analysis Profile (PAP)* is introduced as a domain-specific customization of UML to provide an energy-modeling and, e.g., to integrate the concepts of energy models for UML state machines. For this, the PAP provides additional stereotypes and tagged values to specify, for instance, the electric current consumption and execution time. Since the concept of hardware component models can be mapped to UML elements, e.g., by the aid of the PAP, a seamless integration into the software application model can be performed, resulting in a *system model* which addresses RQ3.

The main contributions are published in [336, 339, 340, 341, 391] and can be summarized as follows:

- A novel system-wide modeling approach based on hardware component models covering *Microcontroller Units (MCUs)* and connected peripheral devices.
- A UML-based description of hardware components that can be combined with a software application model to define a system model. Composed of an energy model as a UML state machine for energy-related non-functional aspects and a UML class for functional aspects, hardware component models provide interfaces for their utilization by software application models in MDD.
- The PAP UML profile to model energy-related aspects. The UML profile is based on the MARTE profile and provides stereotypes to extend hardware component models and introduces new data types to describe the electric current and voltage as NFPs.

RQ4 – Early Evaluation of Energy-aware Software Applications in MDD: *How can the energy-related impact of software application models be determined, and energy-related misbehavior be identified when the hardware platform is not available or only partially available?*

The execution of software application models is based on simulations. Since MDD tools and simulation environments differ strongly w.r.t. their general functionality, an external analysis process has to be developed. For the evaluation of energy-related aspects, concepts to cope with the different conditions in early development stages have to be defined.

To answer RQ4 and overcome *Challenge 4*, the MDD tool used in this thesis is enhanced to transform hardware component models for a tool-independent approach. The simulation environment of the MDD tool has also been extended to allow data exchange with external analysis tools during execution. Moreover, power analysis methods have been developed for the early evaluation of energy-aware software applications in MDD. The main contributions to address RQ4 are published in [339, 340, 341] and may be summarized as follows:

- The *Indirect Power Analysis (IPA)* method provides a simulation-based rapid power analysis with a virtual hardware platform based on hardware models. This method may be used to test software application models without physical hardware, for example, in early development phases.
- The *Direct Power Analysis (DPA)* method, a superset of IPA, defines a novel in-the-loop testing approach and utilizes a physical embedded system (testbed) for the estimation process. To achieve an early power consumption estimation, DPA and IPA are based on the same set of communication protocols and a centralized tool to simulate hardware components (IPA) and to enable communication between the software application model and the testbed (DPA) during simulation. With DPA, software application models executed within a simulation environment can interact with the testbed to obtain, for example, real sensor data.
- A case study of an IoT application for a beehive microclimate sensor node as a proof-of-concept intended to demonstrate the applicability and potential of the overall power consumption estimation approach. The proof-of-concept includes the specification of

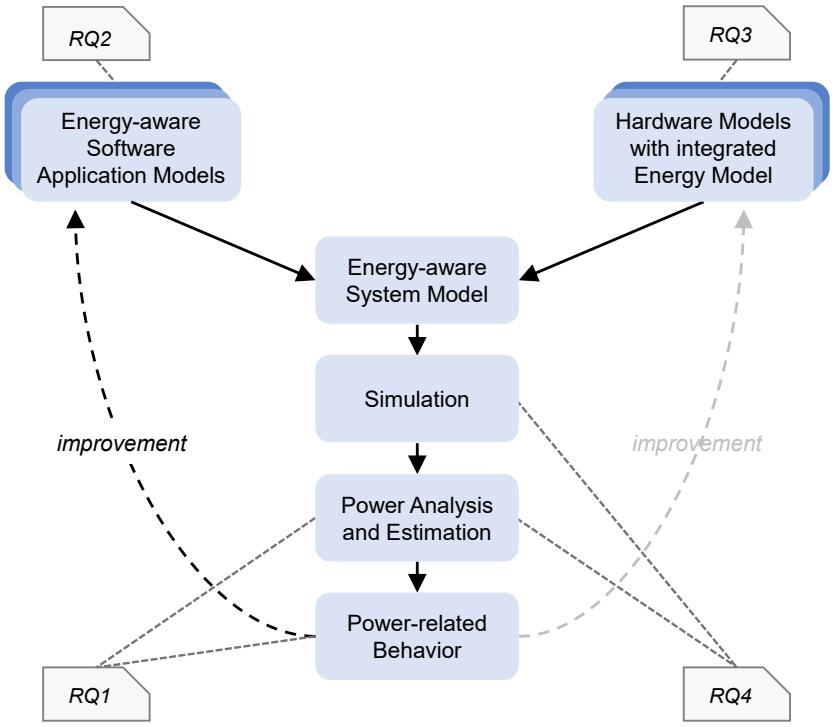


Figure 1.1: Framing and associated RQs of this thesis. Notation according to [197, 198].

software and hardware models, the elaboration of scenarios, and the definition of energy-related NFRs. By this, the provided case study also contributes to the evaluation of RQ1 and RQ3.

Figure 1.1 sketches the overall approach depicted as a Y-chart[2] with associated RQs as notes. For example, contributions to answer RQ1 provide fundamental concepts for the power analysis process, which may be used to improve software application models. Note that the improvement of hardware models may also be achieved but is not in the scope of the presented thesis. Software models may become energy aware when contributions to address RQ2 are applied. Hardware models are defined by concepts resulting from answering RQ3 and are combined with the energy-aware software model into a system model. Contributions to address RQ4 cover parts of the simulation and the power analysis concept to achieve early power consumption estimation.

The introduced contributions to answer RQ1 to RQ4 have been part of different publications. The following Table 1.1 contains a list of core publications addressing the four presented RQs. A complete list of related publications that resulted from this thesis is presented in Chapter *Publications* (p. 243 ff.).

[2]The term Y-chart refers to the shape of the diagram and is not related to the well-known representation to visualize design views and hierarchies introduced by Gajski and Kuhn (1983) [130].

Publication Title	RQ1	RQ2	RQ3	RQ4
As Lead Author				
Framework of software design patterns for energy-aware embedded systems. [337]		✓		
Energy-aware pattern framework: The energy-efficiency challenge for embedded systems from a software design perspective. [338]		✓		
Power consumption estimation in model driven software development for embedded systems. [339]			✓	✓
Towards power consumption optimization for embedded systems from a model-driven software development perspective. [340]			✓	✓
Hunting energy bugs in embedded systems: A software-model-in-the-loop approach. [341]	✓		✓	✓
As Co-Author with equal contributions				
Rapid-prototyping and early validation of software models through uniform integration of hardware [393]				✓
PowerMonitor: Design pattern for modeling energy-aware embedded systems: Work-in-progress. [392]		✓		
Software design of energy-aware peripheral control for sustainable internet-of-things devices. [391]			✓	

Table 1.1: Relationship between core publications and introduced RQs.

1.3 Limitations

This thesis aims to provide novel approaches for the estimation of power consumption as well as the description of energy-aware software design patterns. The introduced concepts of energy bugs, scenarios, and energy-aware software design patterns are specified at a high level of abstraction. Therefore they can be extended, modified, and applied to other domains. However, the power consumption estimation concept has more specific requirements. To provide a proof-of-concept, the presented work is limited in several areas:

- *Type of embedded systems*: We target software application models for embedded systems, especially IoT devices with limited resources and computing power that collect data and send information using specific wireless communication protocols. We specifically do not consider high-performance computing or desktop applications, desktop and server systems, and distributed systems with intensive network communication. The presented concept may be applied to the aforementioned domains. However, no feasibility studies have been conducted on this topic. Additionally, it is not in the scope of this thesis to provide a network simulator.

- *MCU Support*: Another limitation is the support for different MCUs. As an executing platform for the evaluation process, we have developed three prototypes based on the Espressif Systems ESP32 [111], NXP LPC54114 [269], and STMicroelectronics STM32L476 [371] MCUs specifically designed for power measurement and rapid prototyping. The software has been ported for all three platforms. However, the proof-of-concept

presented in this thesis does not aim to provide a direct comparison between or a detailed performance analysis of these MCUs.

- *Modeling Languages*: As the most widely used modeling language in the embedded software industry [7], UML is used to specify models of software applications and hardware components. The concepts presented in this thesis may be applied to other modeling languages providing mechanisms for extending models and describing NFPs. First results for the adaptation of the proposed approach to enable functional testing in MathWorks Matlab/Simulink were submitted as new research and accepted as a scientific publication [393].
- *Quality of Hardware Components Models*: The power consumption estimation process can only be as good as the models used in the estimation process. Generally, the quality of a hardware model depends on multiple factors, e.g., the availability of data sheets, measuring instruments, and the experience of developers. However, the introduced power estimation approach is limited if a hardware component to be modeled has at least one or more of the following properties:
  - *Black Box Model and Behavior*: If hardware components are described as a black box, the power-related behavior can only be deduced from observations, whereby it is not always clear which operations lead to a changed power-related behavior. Additionally, if the current state of the component cannot be queried or determined, the present concept can only assume the expected behavior during a simulation. However, assumptions can lead to significant differences and inaccuracies in the power consumption estimation process.
  - *Environmental Impact*: The presented concepts in this thesis are not designed to include (randomly occurring) environmental effects in the estimation process, leading to a change in the power-related behavior of hardware components. This thesis introduces the concept of scenarios to define a subset of possible environmental impacts for specific use cases. To fully include the effect of the surrounding, an environmental model and simulator or a highly specific laboratory environment are necessary to control, for example, interference in wireless communications or particle changes in the air during simulation. However, such approaches may be coupled with the presented concepts.

  Additionally, power-saving techniques such as *Dynamic Voltage and Frequency Scaling (DVS)* and *Dynamic Voltage Scaling (DVFS)* may, theoretically, also be modeled abstractly but are also not in the scope of this thesis.
- *Code Generation and Optimization*: The thesis aims to estimate power consumption in early development phases. This also includes stages in which the hardware platform is not defined, currently being evaluated, or only partially available. The concepts in this thesis are focused on behavioral and architectural aspects of software application models and not on the characteristics of specific programming languages. However, according to the MDD methodology, the hardware-specific source code of such software applications models is obtained by automatic code generation techniques provided, e.g., by MDD tools. In order to be independent of programming languages, our approach is limited to the simulation of software application models, while source code generation or power-related optimizations on the source code level are not considered.

1.4 Thesis Outline

This section outlines the remaining structure of the thesis. Related topics have been grouped thematically into individual chapters to enhance the readability and the common thread. Unless stated otherwise, the color coding in figures presented in the following chapters enhances the illustration and has no semantic meaning. Figures without references in their caption were developed by the author.

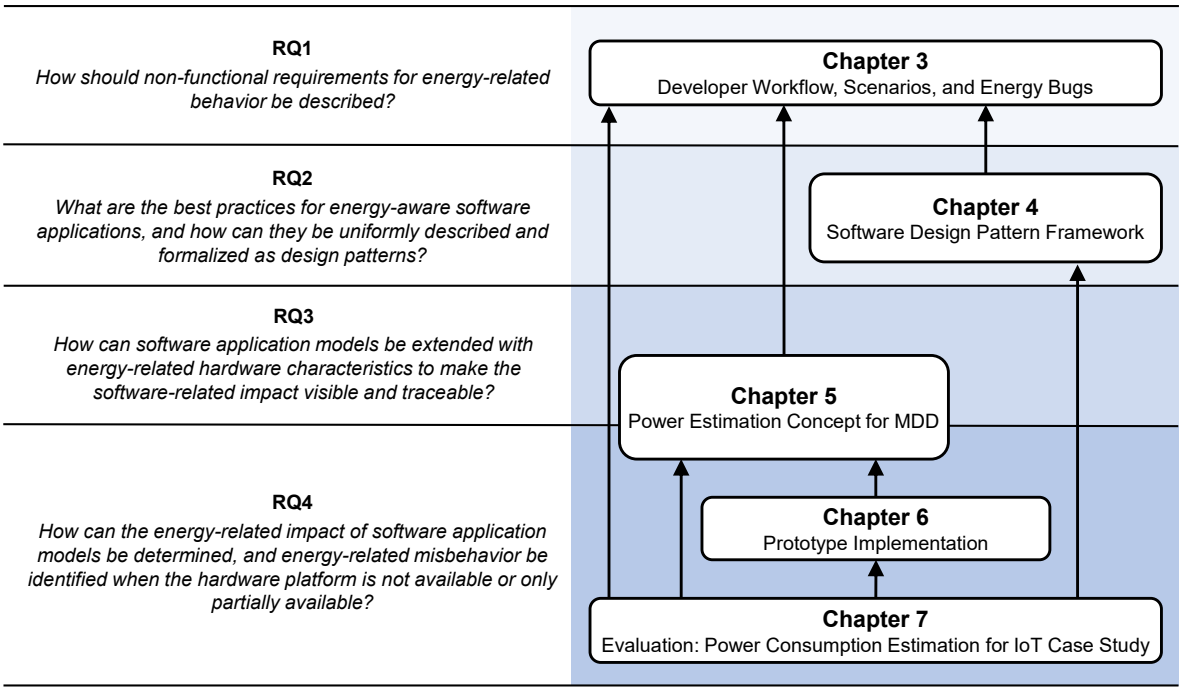


Figure 1.2: Relations between main chapters with addressed research questions. Arrows indicate usage relationships.

Figure 1.2 provides a graphical overview of the relations between the main chapters of this thesis and their associated RQs. Each chapter focuses on one RQ but may also contribute to answering additional RQs. The remainder of this thesis is organized as follows:

Chapter 2 (p. 17 ff.) presents related work and background closely related to the challenges and contributions of this thesis. This includes background in the field of software and systems engineering, including electrical power measurement, embedded systems, embedded software, IoT, requirements engineering, and MDD. Related work is discussed on energy bugs, software design patterns and their correlation to power and energy consumption, the integration of virtual and physical hardware devices, and power consumption modeling and estimation. To further enhance the understanding of methodological, technological, and design choices, this chapter concludes with an additional section to summarize findings and remarks.

Chapter 3 (p. 77 ff.) outlines the overall approach of this thesis and describes how the presented concepts are related and interconnected. In addition, this chapter introduces the basic concepts of scenarios and energy bugs as contributions to answering RQ1, which form the basis for other concepts in subsequent chapters of the thesis.

Chapter 4 (p. 89 ff.) presents the software design pattern framework for identifying and describing energy-aware software design patterns. To overcome RQ2, the framework describes the design pattern identification process and introduces a novel design pattern template. This chapter also provides the first energy-aware design pattern catalog, containing five design patterns reusing existing solutions and one newly developed design pattern. All energy-aware design patterns are uniformly described with the presented design pattern template.

Chapter 5 (p. 115 ff.) introduces the power consumption estimation concept for software applications models in MDD to answer RQ3 and parts of RQ4. In addition to the presentation of the concept vision, this chapter describes the hardware modeling process and introduces a UML profile to model power-related aspects as contributions to address RQ3. To answer parts of RQ4, two power analysis methods are proposed.

Chapter 6 (p. 139 ff.) introduces a prototype implementation of the power estimation methods presented in Chapter 5 as a proof-of-concept. The implementation also includes the development of a policy-oriented *Hardware Abstraction Layer (HAL)*, an external and independent estimation tool, communication protocols, and a set of hardware platforms for the power estimation of system models during simulation.

Chapter 7 (p. 175 ff.) provides the evaluation of the power consumption estimation approach based on the prototype implementation introduced in Chapter 6. A case study of a beehive microclimate sensor IoT node example illustrates the application of the model-driven concepts introduced in Chapter 5. The case study covers the definition of hardware component models in UML for each component of the sensor node and also demonstrates the use of the PAP UML profile to model static and dynamic power-related NFPs. Besides the case study evaluation, the overall performance of the power estimations method with a connected hardware platform is discussed.

Finally, **Chapter 8** (p. 197 ff.) concludes this thesis and summarizes the contributions and findings to overcome the introduced RQs. This chapter also provides a conclusion of this thesis and an outlook with open and new ideas for future research.

Chapter 2

Related Work and Background

This chapter presents the related work and background regarding the proposed concepts and approaches to increase the understanding of this thesis. Since the proposed contributions consider and combine various fields to answer RQ1 to RQ4, the related work and background presented in this chapter cover a wide range of topics. An outline of electrical power measurement fundamentals, metrics, and techniques are presented in Section 2.1. Section 2.2 presents an overview of embedded systems, embedded software applications, and IoT as the basis for the case study presented in Chapter 7 (p. 175 ff.). Aspects of requirements engineering are discussed in Section 2.3. An overview of pattern descriptions and related work on patterns with correlation to power and energy consumption are provided in Section 2.4. Section 2.5 introduces concepts of MDD, while Section 2.6 covers selected aspects of MARTE as a basis UML profile to model non-functional aspects. Section 2.7 gives an overview of software testing principles. Finally, Section 2.9 discusses further findings and remarks, as well as technology and design decisions based on the presented work.

2.1 Electrical Power Measurement

This section presents the physical background on topics related to electrical power, electrical energy, and power consumption measuring techniques. The topics in this section provide basics for modeling power- and energy-related requirements and properties and for measuring embedded systems as part of the power estimation process.

Section 2.1 briefly introduces the terminology and fundamentals of voltage and electric current, as well as power and energy as characteristics of hardware components used, for example, in embedded systems as described in Section 2.2 (p. 21 ff.). Metrics based on the introduced fundamentals, which can be used for requirement evaluation within the concepts presented in Chapter 5 (p. 115 ff.), are covered in Section 2.1.2. Section 2.1.3 discusses different power measurement techniques along with their advantages and disadvantages.

2.1.1 Physical Fundamentals

The first term introduced is the electric charge (Q), which defines a property of matter. It is responsible for a certain force that the matter experiences in an electromagnetic field. The unit of electric charge is defined as coulomb (C). The electric charge can be positive or negative and is carried by subatomic particles, for example, as the elementary charge of an electron which is $\sim 1.602 \cdot 10^{-19}\ C$. An electric field is a structure that surrounds electrically charged

elements. Electric potential refers to the amount of work needed, e.g., energy, to move a unit of electric charge Q from one point in the direction of another point within the electric field. The unit of electric potential is volt (V), which is equal to $J \cdot C^{-1}$, and describes the ratio between potential energy in joules (J) and electric charge in coulomb (C).

Voltage, denoted as U, can be described as the difference in electric potential between two points and, thus, as the quantitative measure of the potential difference. Electric current, also referred to as current in this thesis, is defined as $Q \cdot t^{-1}$ and defines the flow of the electric charge Q across an electrical conductor or space at the rate of one C during an interval of time t. As stated by Ohm's law, the electric current between two points in a conductor is directly proportional to voltage. The electric current is formally denoted as I and measured in ampere (A).

Electric power, denoted as P and also known as power consumption, describes the rate at which an electric circuit transfers electrical energy. For non-constant parts whose characteristics may vary over time, the electric power P is defined as:

$$P(t) = U(t) \cdot I(t) \tag{2.1}$$

The electric power is measured in watts (W), which is equal to $J \cdot s^{-1}$ and $V \cdot A$. The dominant technology used for integrated circuits, such as MCUs and other peripheral devices, e.g., sensors, is *Complementary Metal-Oxide-Semiconductor (CMOS)*. The structure of CMOS-based parts consists of p-type and n-type *Metal-Oxide-Semiconductor Field-Effect Transistors (MOSFETs)* as building blocks to implement internal logic. The two types differ on the doping element used and thus types of charge carriers, such as holes for p-type and electrons for n-type MOSFETs. As a variant of field-effect transistors, MOSFETs use an electric field to control the flow of current. The three pins of a MOSFET are labeled as *Source*, *Drain*, and *Gate*. While the *Drain* is connected to the load, the *Source* is typically connected to a positive voltage for p-type or to ground in the case of n-type MOSFETs. The *Gate* is used to control the flow of current, which has to be a high state for p-type and a low state for n-type MOSFETs to enable current flows. For more detailed information about MOSFETs, the reader may refer to [195, 196, 298]. The power consumption of CMOS-based parts is defined as [157, 196, 297, 425]:

$$P = P_{short} + P_{switch} + P_{static} \tag{2.2}$$

The instantaneous power consumption shown in Equation (2.2) can be subdivided into short-circuit (P_{short}), switching (P_{switch}), and static power consumption (P_{static}). The terms P_{short} and P_{switch} are both related to the dynamic power consumption of CMOS-based components being active and values of signals are changing so that:

$$P_{dyn} = P_{short} + P_{switch} \tag{2.3}$$

The term P_{short} in Equations (2.2) to (2.3) results from the short-circuit current I_{short}, which occurs when the p-type and n-type MOSFETs are simultaneously active during the switching event. In this case, the CMOS conducts current directly from the supply voltage (V_{dd}) to the ground for a short time. P_{short} can be defined as:

$$P_{short} = I_{short} \cdot V_{dd} \tag{2.4}$$

P_{switch} defines the power consumption during the switching activity of the CMOS and is defined as:

$$P_{switch} = \alpha \cdot C_L \cdot {V_{dd}}^2 \cdot f \tag{2.5}$$

with C_L as the load capacitance of the CMOS logic and f as the operating or clock frequency. The activity factor $\alpha \in [0, 1]$ defines the fraction of the circuit that is currently switching [200] or, in other words, the average of the total capacitance of a circuit charged and discharged each cycle at the clock frequency f [195]. The value of P_{short} in Equation (2.4) can be considered as low compared to the power consumption originating from the switching activity because it only occurs for a short time during each transition, which is why P_{short} is often neglected [196]. The static power consumption P_{static} in Equation (2.2) describes a constant power consumption of CMOS-based circuits without any switching activity, which is also called leakage power consumption. In a simplified manner, P_{static} can be defined as:

$$P_{static} = I_{leak} \cdot V_{dd} \tag{2.6}$$

with I_{leak} as the leakage current. However, the definition of I_{leak} is complex since various effects lead to different types of leakage, including sub-threshold leakage, gate leakage, reverse-biased-junction leakage, gate-induced-drain leakage, gate-oxide leakage, gate-current leakage, and punch-through leakage [157, 195, 196].

Energy defines the ability to do work. Electric energy E is measured in Joule (J) with $J = V \cdot A \cdot s = W \cdot s$ and is related to the fraction of potential energy required to move a charged element Q between two points in an electric field. As defined in Equation (2.7), electric energy can be calculated by considering the rate of energy transfer (electrical power) over a given time.

$$E = \int_t P(t)\ dt \tag{2.7}$$

2.1.2 Metrics

If questions about the performance and characteristics of a system have to be answered, metrics come into play. Generally, a *metric* can be defined as a function used to map a property or characteristic of a system to a numerical value, denoted as a measure. This measure can be used as a performance indicator to achieve comparability, e.g., to evaluate the quality or effectiveness of improvements, to identify the most suitable solution, and to decide if functional and non-functional requirements have been fulfilled. In addition to metrics, methods may be defined describing how to perform the investigations and measurements from which the measures are derived. The two basic metrics of power P and energy E (cf. Section 2.1.1, p. 17 ff.) may be used for the power consumption estimation concept presented in this thesis.

With P as the metric for power, the peak and average power consumption of a system may be measured and used to predict properties related to heat dissipation and cooling, for instance. In addition, this metric may also be used to express power supply constraints and maximum battery loads. Using the energy consumption E as a metric allows an estimation of the battery consumption and expected lifetime of the system or the dimensioning of critical components such as batteries.

Metrics can also be combined with other metrics and properties of the system in order to define more complex metrics, e.g., for CMOS circuits. The power-delay product, for example, is an indicator of the energy consumed per switching event and is defined as the product of the average power consumption times the input-output delay, propagation delay, or duration of the switching event. The energy-delay product is another metric in the CMOS domain for energy and performance which is specified as the product of the power-delay product times

the delay. However, the concept of the energy-delay product has been adapted in the past, e.g., as an indicator of the efficiency of programming languages [136] or high-performance computing applications [363]. It might also be suitable for the consideration of the load on the battery. Additionally, different variants of the energy-delay product exist where single factors are potentiated by integer values and, thus, introduce a weighting to increase the significance of either the energy or the execution time.

The basic metrics are sufficient for estimating power consumption as described in Chapter 5 (p. 115 ff.). It should be noted that the definition of energy-related NFRs (cf. Section 2.3.3, p. 30 ff.) and the resulting concept of energy bugs (cf. Section 3.3, p. 84 ff.) are not limited to specific metrics and may be extended to consider additional or different metrics, such as the power-delay product.

2.1.3 Measurement Techniques

Different techniques exist to measure the electric current consumption of a *System Under Test (SUT)* consisting of a single hardware component or a full-fledged embedded system [128, 418, 428]. Resistance-based and charge-based measurements are two of the most common principles for measuring direct current in an electric circuit. Alternative approaches based on current transformers, Rogowski coils, or Hall-effect sensors that measure current indirectly via magnetic fields or inductions are neglected because their underlying physical principle may only work in alternating current circuits, may be used in much higher current scenarios, or may have low accuracy.

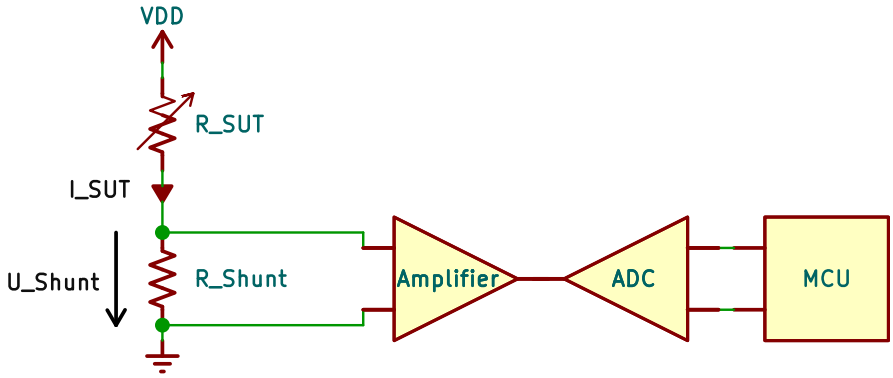


Figure 2.1: Schematic for a shunt-based current measurement. Notation based on [169].

For the resistance-based current measurement, a resistor with low resistance, also called a shunt resistor, is used in parallel with an ammeter or multimeter. The measured voltage across the shunt resistor, also referred to as voltage drop, is proportional to the current. Figure 2.1 shows a simplified circuit design for shunt-based measurements. By using Ohm's Law, the actual electric current value can be calculated as follows:

$$I\_SUT = \frac{U\_Shunt}{R\_Shunt} \tag{2.8}$$

The SUT is considered as a variable resistance in the circuit diagram shown in Figure 2.1. A shunt resistor is placed in series to the SUT so that the current which passes the SUT also has

to flow through the shunt resistor. As indicated in Figure 2.1, an amplifier and an *Analog-to-Digital Converter (ADC)* are typically used to sample the current value. A high resistance of the shunt resistor results in a greater voltage drop, and by this, effects of the circuit, e.g., due to static and thermal noise, will be minimized, and the overall measurement will be improved. However, a high voltage drop negatively affects other components in the circuit. Based on the expected electric current to be measured, selecting the best suitable shunt resistor is a tradeoff between measurability and minimizing negative effects. The voltage drop can be addressed if the measuring device is connected to a regulated power supply in a closed-loop manner to dynamically adjust the voltage powering the device. In this configuration, it can be ensured that the SUT always receives the intended operating voltage [240]. For a fine-grained current measurement, a high sample rate is required. Because this technique is based on sampling, values between two samples remain unnoticed. As the time difference between two samples increases, more data, such as peaks in both directions, cannot be sampled, making the measurement less accurate.

A technique that solves the sampling problem is denoted as charge-based measurement. For instance, a concept presented in [160] uses a transistor-based current mirror circuit designed with two capacitors. The basic principle of this technique is that the transistor mirrors the current drawn by the SUT while a capacitor, acting as a short circuit, is charged. As exemplary concepts presented in [160, 206], a RS flip-flop switches between the capacitors if one is fully charged and initiates a discharge while the next capacitor starts charging simultaneously. Since the capacitance and the voltage are known, the power consumption can be calculated by counting the charge-discharge events as another form of sampling. While sampling problems do not directly affect this technique, other disadvantages exist. First, the transistor's characteristics must match the SUT's characteristics in terms of the electric current consumption range. Due to the increased complexity of the current-mirror circuit design, adaptions are harder to achieve compared to the introduced resistance-based technique. Second, the capacitors have to be scaled so that they are small enough to get a high resolution but still be able to collect sufficient data since the capacitor is not collecting any charges while discharging. If the capacitance is chosen too large, this approach may be unable to locate peak currents time-wise. When neglecting the switching time, this technique is able to accurately measure the average current used by the SUT.

2.2 Embedded Systems

It is expected that over 95 % of software applications developed are aimed to be executed on embedded systems [285]. As a *"combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function"* [132], embedded systems have become ubiquitous and can be found in domains such as automotive, aerospace, agriculture, robotics, consumer electronics, healthcare, IoT, and IIoT [36, 146, 261]. This section introduces background on embedded systems as a target platform for the presented early power consumption estimation of software applications in MDD. Section 2.2.1 introduces the architecture, characteristics, and challenges of embedded systems, while Section 2.2.2 focuses on software applications for embedded systems and describes their impact on power consumption. Section 2.2.3 introduces IoT as the field of application for embedded systems considered in this thesis.

2.2.1 Architecture and Characteristics

This section discusses the architecture of embedded systems from a hardware perspective, their characteristics, and the typical challenges such systems have to overcome. Figure 2.2 illustrates the generic architecture with typical components of an embedded system consisting of a *Central Processing Unit (CPU)*, a fixed amount of volatile and non-volatile memory such as *Random-access Memory (RAM)*, flash memory, and *Read-only Memory (ROM)*, *General Purpose Input/Outputs (GPIOs)*, ADCs, and *Digital-to-Analog Converters (DACs)*. In modern embedded systems, MCUs are used that combine the aforementioned components within a single integrated circuit and may include other serial communication interfaces such as *Serial Peripheral Interface (SPI)*, *Universal Asynchronous Receiver Transmitter (UART)*, and *Inter-Integrated Circuit (I^2C)* [298]. Embedded systems, such as the example in Figure 2.2, are heterogeneous and distributed by design. Different types and numbers of MCUs can be combined into a *System-on-Chip (SoC)*. An embedded system may also consist of various sensors, e.g., measuring temperature, humidity, light, pressure, audio, or gas, and actuators interacting directly with the environment, such as displays and motors. Moreover, wireless and wired communication interfaces enable interaction with other systems.

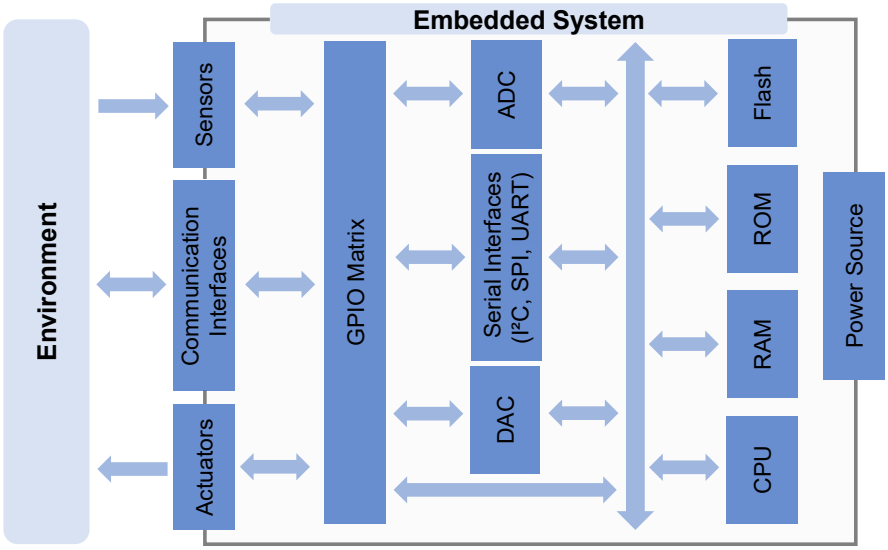


Figure 2.2: Generic block diagram of an embedded system architecture. Horizontal arrows describe bi-directional communication between components, while the vertical arrow represents a communication bus.

Compared to general-purpose systems, for instance, workstations designed to interact with other devices and users to perform various tasks, an embedded system is intended to be used for a limited but highly specific task while interacting with the physical environment [35]. Embedded systems have to be fault tolerant, robust, and resilient to spontaneous events, condition changes, and extreme conditions such as humidity, temperature, or radiation of the environment in which they are placed [59, 222]. As an additional characteristic, they are required to perform the intended tasks without maintenance, have restricted resources in terms of processing power, memory size, and operating system support, as well as a large number of constraints, including real-time, efficiency, safety, costs, weight, size, reliability, and

Class	Data Size	Code Size
Class 0	$\ll$ 10 KiB	$\ll$ 100 KiB
Class 1	$\sim$ 10 KiB	$\sim$ 100 KiB
Class 2	$\sim$ 50 KiB	$\sim$ 250 KiB

Table 2.1: Classification of restricted devices based on RFC 7228 [51].

energy consumption as one of the most important constraints [240, 261, 278]. A definition and classification[1] of constrained devices is provided by RFC 7228 [51] and shown in Table 2.1. It consists of three classes for devices with much less than 10 KiB data size and 100 KiB code size (Class 0) to devices with 50 KiB and 250 KiB for data and code size (Class 2).

Embedded systems can also be distinguished, for instance, by the power source (Figure 2.2), which may be a static power supply connected to a main or a battery that may or may not be rechargeable. As a categorization, the RFC 7228 also defines classes of energy limitation and general power usage strategies, such as always-on, normally-off, and low-power devices [51].

Considering the characteristics of an embedded system, various factors, such as thermal effects, can lead to energy awareness for hardware components and software applications [240]. Due to shrinking processes of the circuits and reduced transistor threshold voltages, static power consumption (cf. Section 2.1.1, p. 17 ff.) has gained importance and is now responsible for up to 40 % of the total power consumption [297, 425]. Higher temperatures lead to increased power consumption while reducing the reliability of the embedded system. Imagine, for example, size or cost constraints that prevent the use of active and passive cooling systems. To deal with such design constraints, it might be necessary to lower the frequency (clock gating) or the supply voltage (power gating) [195] for the hardware components such as MCUs or SoCs. Power gating, for instance, might lead to a quadratic decrease of the dynamic power consumption (cf. Equation (2.5), p. 18). Specific segments and functional blocks can be powered down entirely to address the static power consumption. Phenomena such as the utilization wall [405] and the power wall [297] refer to these limitations from a slightly different perspective. The utilization wall is closely related to the definition of dark silicon [107, 194, 240]. Both refer to the number of functional blocks that can or can not be powered simultaneously for a given thermal design power constraint. The power wall is quite similar and refers to power density and heat dissipation limits which have reached the practical power limit for cooling systems.

However, if energy is a limited resource, the selection of hardware components based on power consumption and the software application, as the behavior of the embedded system, is becoming more important when building power-efficient battery-operated embedded systems [285]. As shown in Table 2.2, the MCU may not be the main consumer within an embedded system. For the power consumption estimation, it is therefore insufficient to consider MCUs without peripheral devices, the intended use case, and the software application that controls and directs most hardware activities.

[1] The classes have been defined based on commercially available products in 2014. The author expects that the borders have to be adjusted to cover today's standard. However, the RFC 7228 has not been updated since.

Device	Cur. Consumption	Voltage	Power
NXP LPC54114 MCU [269] (Cortex-M4 & Cortex-M0+ @ 96 MHz)	Active: 9.9+8 mA Sleep: 3.0 mA	3.3 V 3.3 V	59,07 mW 9.90 mW
STMicroelectronics STM32L476RG MCU [371] (80 MHz)	Active: 10.2 mA Sleep: 2,96 mA	3.3 V 3.3 V	33.66 mW 9.77 mW
ams Osarm TSL25911 Ambient Light Sensor [17]	275 µA	3.0 V	0.875 mW
Bosch BME280 Environmental Sensor [53] (Indoor Navigation)	633 µA	1.8 V	1.14 mW
Bosch BMM150 Geomagnetic Sensor [52] (High Accuracy Preset)	4.9 mA	2.4 V	11.76 mW
Kingbright AP2012EC Red LED [201] (0805)	20 mA	2.0 V	40.00 mW
Melexis MLX90640 Infrared Thermal Sensor [243]	20 mA	3.3 V	66 mW
Microchip ATWINC15x0-MR210xB 802.11 b/g/n IoT Module [250] (802.11g 6 Mbps code rate)	291 mA	3.3 V	960.30 mW
Plantower PMS1003 Particle Sensor [302]	100 mA	5 V	500 mW

Table 2.2: Exemplary hardware devices with their average energy characteristics.

2.2.2 Embedded Software

The amount of embedded software has grown faster than Moore's law [285] and makes up a significant part of the embedded system engineering. At a consistently high level over previous years, around 60 % of the resources, e.g., time, costs, and developer efforts, are spent in the software development process [26]. The need for more extensive technologies, algorithms, protocols, and standards, as well as the increasing amount of software required, makes the development of software applications for embedded systems a complex and challenging task. Embedded software is typically developed at a low level of abstraction, while programming languages such as C, C++, or *Assembly (ASM)* are used to gain direct control over specific hardware components [26, 285, 298]. Software is also a key component in any embedded system that must be fault-tolerant and implement concepts for handling hardware failures. Due to this, the software application has to be platform specific to meet high-performance requirements while dealing with limited resources. Still today, the software is involved in many serious failures of embedded systems [162, 202].

In addition to functional requirements, embedded software applications are required to fulfill important and restrictive NFRs, such as reliability, real-time, and power consumption, among others described, e.g., in [332]. Note that for embedded systems in specific domains, the number of NFRs the system has to satisfy exceeds the number of functional requirements [247]. Reliability, for instance, defines the probability that a system works as intended. Since the software application is affected by the non-deterministic physical environment and may react to spontaneous events and condition changes [35, 36], reliability is important in the embedded domain. Soft or hard real-time NFRs [285] are especially important for safety-related systems if, for instance, the execution time between data sampling and analysis and the reaction of the system should not exceed a predefined time limit. Power consumption is an important aspect that can be a critical bottleneck for embedded systems [35]. A violation of such requirements can lead to disastrous consequences.

The software application itself does not consume any power. However, the power and energy consumption of an embedded system can be interpreted as the result of the software application's execution and behavior. This connection can also be interpreted as a cause-effect relationship between the software level and the hardware level [160]. Actions taken by the software application, for instance, executing commands on the MCU or interacting with a sensor to obtain readings, are causing effects, e.g., increased power consumption, on the hardware component due to the execution of the requested action. As mentioned in [160], improved hardware designs aim to lower the static portion of the power consumption (cf. Section 2.1, p. 17 ff.). In contrast, the dynamic power consumption caused by the software application during runtime may be optimized on the software level. However, while this statement is valid and the software level cannot change the physical properties of electronic parts, it is worth mentioning that on a higher view, the software application might be able to power off hardware components at exact points during execution to lower the overall static power consumption. To address the power consumption of individual hardware components, software applications may affect the following parameters as part of the introduced equations in Section 2.1 (p. 17 ff.):

- *Voltage*: By adjusting the voltage of the system and single components, the power consumption may be lowered. This concept is also called DVS and DVFS [65, 223, 300]. Alternatively, the power from separated parts of the system can be turned off, which requires hardware layer support. Software applications do not always have control over those features.

- *Frequency*: The software application may adjust the operating frequency of components, e.g., the MCU, in particular situations to reduce power consumption. Techniques such as *Dynamic Frequency Scaling (DFS)* and DVFS [300] are not or only partially supported by low-cost and low-end MCUs like the ARM Cortex-M family, which are often used in battery-powered embedded systems. On a higher abstraction level, software applications can also adjust the sample rate to reduce the power consumption of specific sensors.

- *Capacity*: Functional units can be completely disabled to lower the power consumption of a component and avoid thermal issues (dark silicon). Strategies must be implemented to control the active states of hardware components and functional units.

- *Time*: The time a system or component operates in active mode with high power consumption can be minimized if the software application's workflow is optimized, and effective algorithms are used.

The growing complexity of embedded software applications leads to increased software-based quality problems in the industry [222] and, thus, the non-fulfillment of quality attributes, expressed as functional and non-functional requirements [14, 137]. To manage the complexity and shift the focus of developers on the essential complexity, MDD, as introduced in Section 2.5 (p. 37 ff.), has been used for decades as a development methodology for software applications in the embedded domain [222, 244, 349]. In MDD, a model representation of the software application as a higher level of abstraction is used. The methodology also copes with the ability to use code-generation approaches to derive source code from models to native programming languages such as C and C++. Therefore, MDD can increase the software quality due to more expressive notations and automation while reducing the overall development costs and the time-to-market.

2.2.3 Internet of Things

The *Internet of Things (IoT)* is an important and one of the fastest-growing technologies where embedded systems play a significant role [26]. The actual estimation of active devices and the annual growth rate may vary depending on the type of research and institution. As shown in Figure 2.3, starting from 2022 with 13.2 billion devices, [257] predicts almost a threefold increase to more than 34.4 billion active IoT devices in 2032, a compound annual growth rate of 10 %.

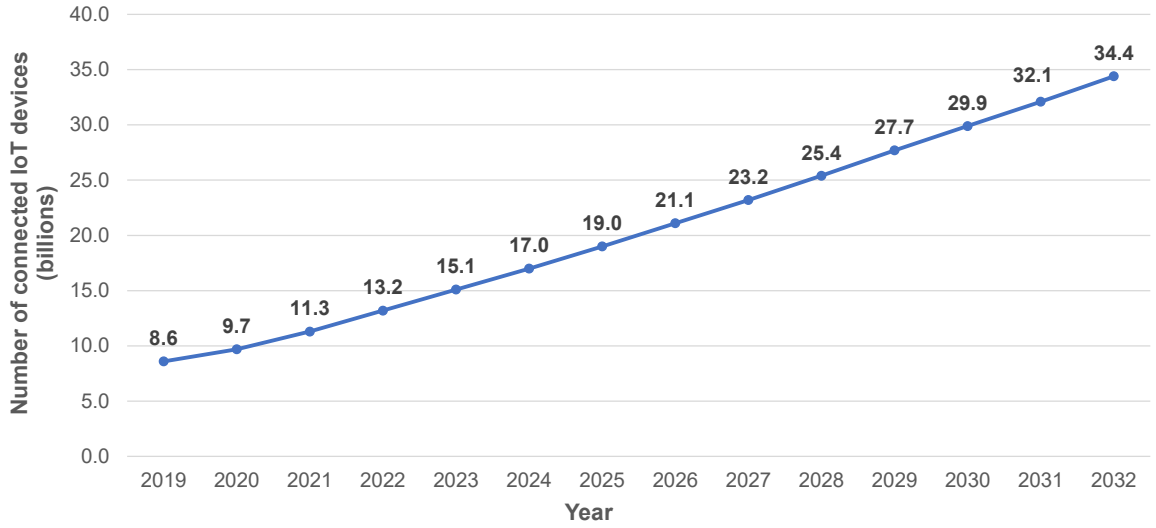


Figure 2.3: Number of worldwide active IoT devices in billion from 2019 to 2022 and forecasts to 2032, adapted from [257, 401].

The term IoT does not have a universally accepted definition and is interpreted differently in research and industry depending on the view and scope, for example, w.r.t. domains, devices, platforms, and communication protocols [47, 142, 354]. However, common to most of the significant definitions, the concept of IoT describes a network with the ability to connect smart objects which can interact with the environment, humans, and other smart objects and compute and share information across platforms. As a key technology of Industry 4.0 [215], IIoT [333] describes a subset of IoT in which solutions are used in industrial environments, e.g., for manufacturing machines in production facilities [218]. A broad overview of the application domains of IoT is pictured in Figure 2.4. It contains domains like smart city, smart agriculture, healthcare, and manufacturing, which can be further subdivided into various subcategories as proposed, for example, in [28, 218]. One of the main goals of IoT and IIoT is the transformation of objects, such as consumer products, machines, or plants, into smart objects. The concept of smart objects was introduced by Kallmann and Thalmann (1999) [192] as a definition for objects able to describe their possible interactions with virtual human agents in a virtual world. However, smart objects have become a key component in the IoT concept [208]. They are defined as physical objects with unique identities and decision-making capabilities that can sense the environment, store data, and communicate autonomously with other participants.

The transformation of an object into a smart object is typically realized using embedded technology, as introduced in Figure 2.2 (p. 22). Such embedded systems are considered

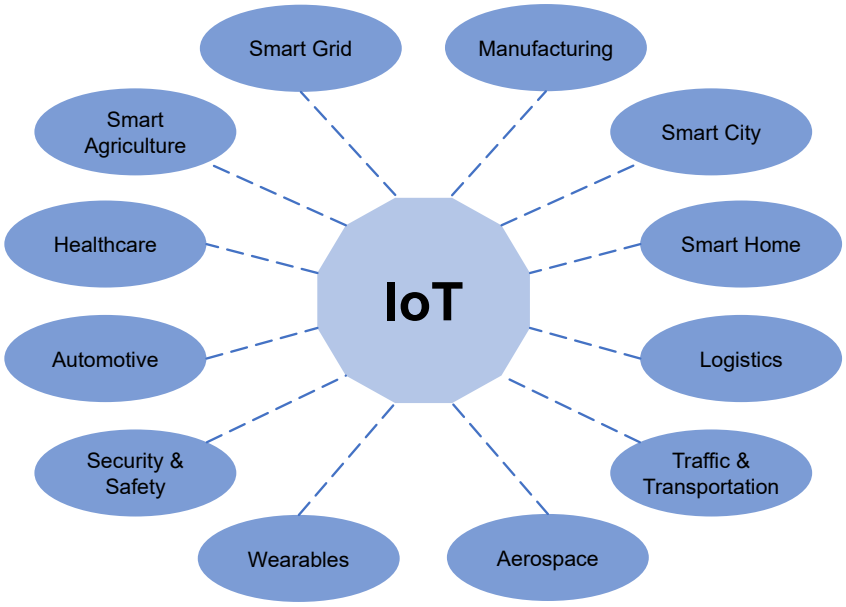


Figure 2.4: Common application domains of IoT [28, 187, 218, 257].

particularly constrained and have a stronger focus on security and privacy. Additionally, constraints in size may result in smaller batteries being used. As a result, the embedded system must be operating even more energy-efficient to reach the expected lifetime. Especially the power consumption of wireless transmission interfaces often exceeds the power consumption of the remaining components of the embedded system (cf. Table 2.2, p. 24). Imagine a battery-powered embedded system that is inaccessible, for instance, buried in the soil, as described in [146]. Specific *Low Power Wide Area Network (LPWAN)* protocols for small data volumes and large ranges have been developed to achieve a long battery life. For instance, *Long Range Wide Area Network (LoRaWAN)* [230] and *Narrowband Internet of Things (NB-IoT)* [1] are frequently used in IoT [256, 415].

This work refers to IoT concentrates on areas where embedded systems are utilized, such as smart devices. A more in-depth description of the other main parts included in the IoT ecosystem, such as technologies, platforms, infrastructures, cloud solutions, and applications of IoT, may be found in [25, 76, 218, 219, 240].

2.3 Software Requirements

This section introduces requirements as an essential part of software development and testing. In this thesis, requirements are used to document non-functional aspects related to electric current, power, and energy consumption. A definition and a classification of requirements in software engineering are introduced in Section 2.3.1. Section 2.3.2 discusses *Non-functional Requirements (NFRs)* as the main type of requirements used in this thesis. Section 2.3.3 covers the energy-related misbehavior of a system, which can be detected if corresponding NFRs are violated during testing. In addition to documentation, the analysis and validation of requirements are further steps in the requirements engineering process covered by software testing, which is discussed in Section 2.7 (p. 49 ff).

2.3.1 Overview

Requirements define capabilities that the system should meet or conditions under which the system should operate. For embedded systems, requirements may be based on electric current, power, and energy consumption. Therefore, energy-aware design patterns (cf. Chapter 4, p. 89 ff.) may be applied to the software application to satisfy a given NFR. In addition, energy bugs (cf. Section 3.3, p. 84 ff.) may also be detected if corresponding NFRs are violated.

Requirements engineering describes a systematic approach in the software engineering process consisting of the four main activities of eliciting, documenting, validating, and managing requirements [303]. A requirement itself can be defined as a *"provision that contains criteria to be fulfilled"* or, in more technical terms, as a *"condition or capability that must be met or possessed by a system, system component, product, or service to satisfy an agreement, standard, specification, or other formally imposed documents"* [174]. According to Pohl and Rupp (2015) [303], elicitation describes the process of obtaining requirements from stakeholders and other sources and refining requirements in greater detail. Validation refers to the validation process of requirements, while management refers to structuring and preparing requirements so that different stakeholders can use and interpret them. The scope of this thesis is limited to the documentation and validation activities, and it is assumed that the requirements are already defined by other stakeholders earlier in the software development process. A common classification approach of requirements in software engineering [303] is the division into functional requirements, NFRs, and constraints, as illustrated in Figure 2.5. According to Sommerville (2016) [362], requirements can be defined more generally on a higher

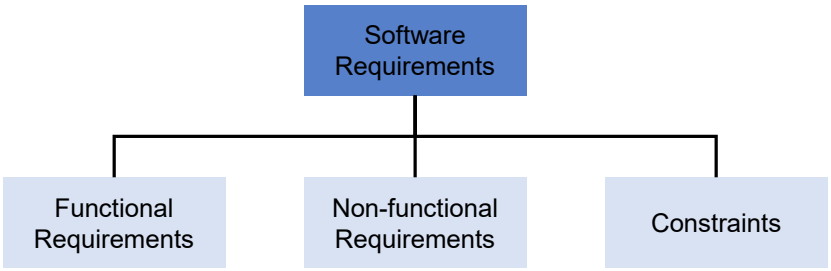


Figure 2.5: Classification of requirements in software engineering.

level, denoted as user requirements, and more detailed on a lower level in the form of system requirements. Functional requirements define the behavior and functionality of the system and, thus, describe the most intuitive requirements for software developers on which most testing approaches focus. An example of a functional requirement may be: *The system shall be able to assign a unique identifier to each temperature measurement of the environment.* Constraints may not be part of the software application and are defined for the system or software development process. The main purpose of constraints is to limit the solution space, for instance, by defining a constraint such as *the software should be implemented using C++* to restrict development to a specific programming language. NFRs[2], as the third class of requirements shown in Figure 2.5, are essential in the context of this thesis and are introduced in more detail.

[2]In the literature, NFRs are also defined as quality requirements, e.g., in [137, 303, 332] and extra-functional requirements, e.g., in [284, 290].

2.3.2 Non-functional Requirements

Instead of a generally accepted and used definition, different definitions of the term NFRs can be found in the literature [137]. A definition that corresponds to the context of the thesis is given in [185], where NFRs specify *"[...] system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability "* and additionally *"[...] physical constraints on a functional requirement"*. Where functional requirements address the system's behavior and, thus, targeting on what the system has to do, NFRs, according to the provided definition, refer to how the system should perform w.r.t. observable attributes [214]. Such observable attributes are denoted as quality attributes. Therefore, a system must have certain quality attributes for which NFRs define the evaluation criteria. However, there exists no unified description for quality attributes. Instead, as described in [235], several software quality models exist to define, group, and

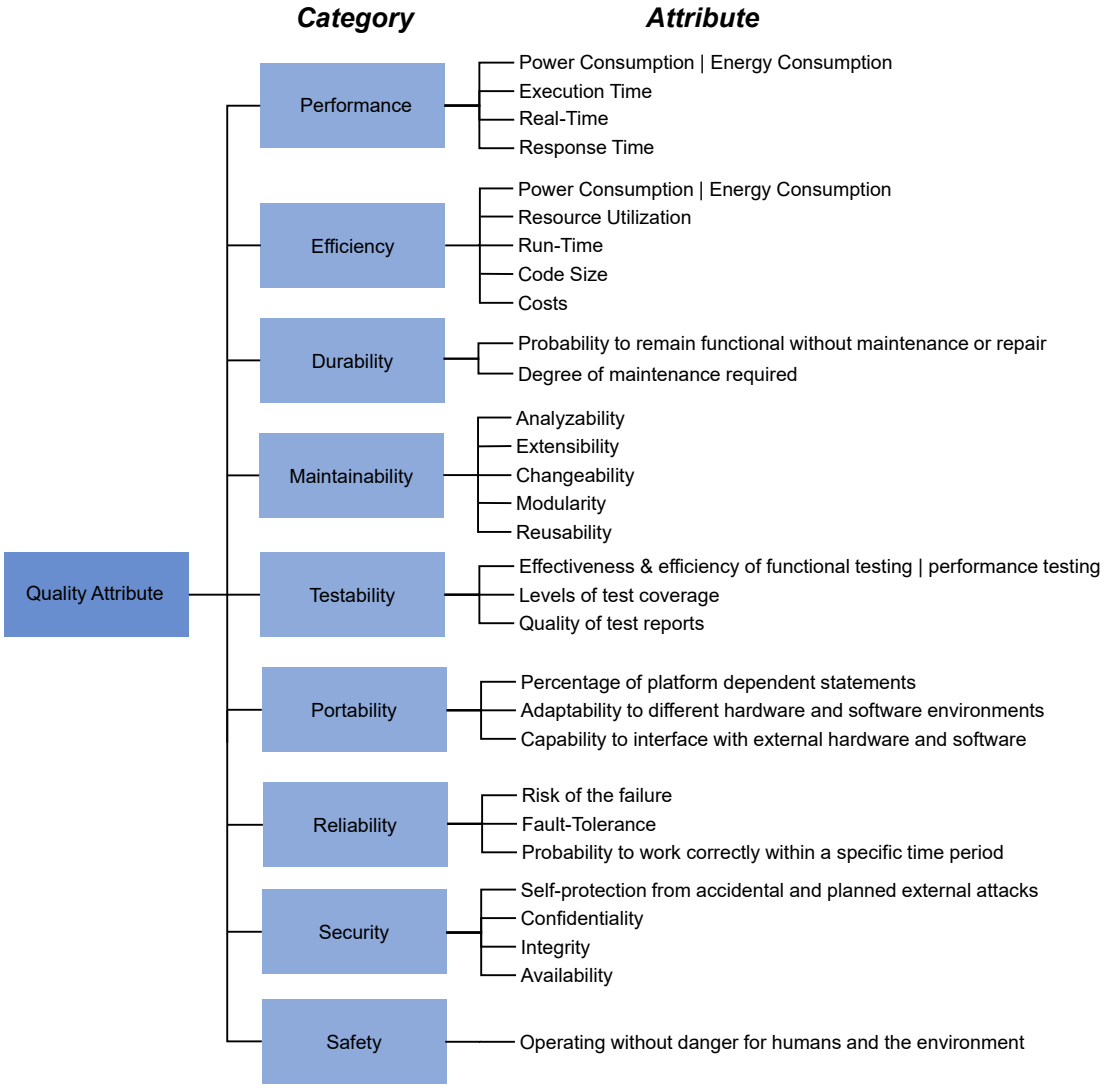


Figure 2.6: Categories of quality attributes to define NFRs [36, 235, 332, 362].

describe the characteristics or quality factors of software applications. Figure 2.6 illustrates typical categories of quality attributes used to define NFRs.

In general, non-functional behavior refers to the aspects of the SUT that are not directly related to its primary function but rather to its performance, reliability, and other quality attributes illustrated by Figure 2.6. However, depending on the nature of the problem and the complexity of the SUT, detecting non-functional misbehavior may be challenging and requires domain knowledge, specialized tools, and techniques. It also may be harder to detect and reproduce non-functional misbehavior than functional misbehavior since complex interactions between different components of the SUT and factors such as hardware configuration or system workload may be involved. The power consumption level, for instance, may not affect the functional behavior of a mains-powered SUT. However, battery-powered SUTs may suffer from high power consumption leading to downtimes or errors, directly affecting the system's functionality. Therefore, it is crucial to use NFRs to specify the expected non-functional behavior as accurately as possible.

Quality attributes considered in this thesis are related to the performance, efficiency, and reliability aspects of the system. Power and energy consumption are generally related to performance and efficiency NFRs in systems engineering:

- As a performance requirement, power and energy consumption directly affect the ability of a SUT to function effectively. For instance, the power consumed by an embedded system SUT may affect the response time or processing speed and, thus, the overall performance.

- As an efficiency requirement, power and energy consumption are concerned with how efficiently the SUT uses this resource to perform its tasks. By reducing power consumption, the SUT may operate for longer periods or generate less heat, which also contributes to the overall system efficiency.

In [207], power and energy consumption are also associated with security issues, for instance, when attackers infiltrate a battery-powered embedded system to cause system failures by attempting to drain the battery. Terms such as *software energy consumption* and *energy consumption of software* [18, 289, 301, 360] indicate that power- and energy-related requirements can also be defined for software applications w.r.t. the utilization of hardware resources and behavior-related aspects. The impact of software applications on power consumption has been discussed in-depth in Section 2.2.2 (p. 24 ff.).

2.3.3 Related Work in the Field of Energy-related Misbehavior

This section presents related work in the field of energy-related misbehavior that has been considered during the elaboration of the novel energy bug and scenario definitions described in Chapter 3 (p. 77 ff.) to answer RQ1. As illustrated in Figure 2.6, power and energy consumption can be both a performance and efficiency quality attribute of the SUT. The violation of corresponding NFRs may indicate the existence of energy-related misbehavior.

The behavior of a system to consume more energy than required to fulfill the intended task is already known. The term energy bug to describe such behavior has gained more attention due to the increasing number of battery-powered embedded systems over the last few years. Researchers have published numerous works [83, 101, 221, 424] to analyze energy consumption and to detect and describe energy bugs for smartphones and mobile applications [34, 294, 293].

Besides energy bugs, energy hotspots, as a further term, are often mentioned when considering the energy-related behavior of software applications. However, the term is interpreted differently by researchers. In [264, 265], energy hotspots describe parts of the software application where (the most) energy is being consumed. The authors of [264, 265] focus on Java-based applications for general-purpose systems. For such a definition, it is important to note that energy hotspots may not be understood as bugs but may indicate abnormal functional behavior. In [34], an energy hotspot for mobile systems, e.g., smartphones, is described as a scenario where the software application causes the systems to consume an abnormally high amount of battery power, even if the utilization of hardware resources is low. This kind of behavior is more related to energy bugs. The first taxonomy of energy bugs is presented in [293]. It distinguishes between hardware-related (e.g., faulty battery and damaged hardware), software-related (e.g., no-sleep bugs and configuration changes), external (e.g., wireless signal strength), and unknown types of energy bugs. Another type of bug is denoted as an energy leak bug [226] describing the cost-ineffective use of peripheral devices or network data.

In [295], the no-sleep bug as a variant of an energy bug is analyzed through a static source code analysis. The identified main causes, e.g., programming errors and higher-level conditions, prevent the system from entering a lower power mode. Wakelocks, a type of energy bug common in mobile software applications, are analyzed in [8, 227, 296]. A wakelock describes the behavior of the software application or the underlying operating system to keep certain hardware components in an active operating state, even if they are currently not being used, leading to unnecessary power consumption.

In [34], a classification of energy bugs and energy hotspots for smartphones is presented. The classification includes four different classes:

1. hardware resources
2. sleep-state transition heuristics
3. background services
4. defective functionality

Each class can be divided into energy bugs and hotspots describing resource leaks, wakelocks, and suboptimal resource bindings. However, the proposed definition of an energy bug is limited to scenarios in which a malfunctioning application prevents parts of the system *"[...] from becoming idle even after it has completed execution"* [34]. Banerjee et al. (2014) [34] also measure the energy inefficiency of an application by introducing an energy-consumption-to-utilization ratio. A high ratio during the execution of the software applications defines an indicator for an energy hotspot. In contrast, a persistently high ratio, even after the software application has completed the execution, is interpreted as the presence of an energy bug.

Related work considering energy bugs discussed in this section provides an informal description in a natural language while focusing more on their effects rather than their causes. Instead, this thesis follows a different approach. To address RQ1, the energy-related behavior of the SUT is formally defined by the use of energy-related NFRs. The non-compliance of the defined energy-related NFRs during testing leads to the derivation of energy bugs, for which a more comprehensive categorization will be introduced. Furthermore, related work does not consider the environment as an essential factor affecting the energy-related behavior of embedded systems, which will be addressed by further concepts presented in Chapter 3 (p. 77 ff.).

2.4 Software Design Patterns

This section presents background on design patterns related to the definition of the novel framework for describing energy-aware software design patterns introduced in Chapter 4 (p. 89 ff.). In Section 2.4.1, pattern formats and pattern classifications are discussed. Power- and energy-related aspects in design patterns as related work are presented in Section 2.4.2.

2.4.1 Formats and Classification of Patterns

This section briefly discusses formats and descriptions of software design patterns while focusing on the form of presentation instead of single design patterns or design pattern characteristics.

Christopher Alexander first introduced the term *design pattern* in [10] as a concept that aims to describe a *"problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*. As problem-solution pairs, design patterns describe best practices for particular recurring design problems. They provide a well-proven generic scheme for their solution of the design problem, typically originating from design knowledge gained by experienced practitioners [67]. With the first catalog of well-described design patterns, the work proposed by Gamma et al. (1994) [131] has become one of the most common and widely accepted sources of software design patterns in software engineering of object-oriented software applications. However, design patterns and design pattern catalogs have been defined for several sub-domains related to the engineering and design of software and systems, including cloud computing [121], human-machine interaction [40, 50, 386], security [345], enterprise applications [124], software architecture [67, 68], messaging systems [159], real-time systems [98], embedded systems [99], or programming language-specific design patterns [41, 82]. Various key aspects may be identified to define design patterns uniformly. This section further presents formats to describe patterns and discusses classifications of design patterns based on their granularity, level of abstraction, scope, and purpose.

Pattern Formats

A pattern format[3] is a uniform and detailed structure in which essential elements of a design pattern can be described. Such a uniform description of patterns within the same catalog or domain enhances the understanding of developers and engineers. It also enables a direct comparison between patterns, e.g., to identify alternative solutions for a design problem. As stated in [10, 67], a pattern consists of the elements *context*, *problem*, and *solution*. By this, a pattern construct creates a relationship *"between a given context, a certain problem arising in that context, and an appropriate solution to the problem"* [67]. In [131], patterns are defined by the four essential elements of a pattern name, a problem description, a solution description, and consequences to describe tradeoffs. Since no domain-specific or universal pattern format exists, several domain and author-specific representation formats have been proposed in the previous decades. However, certain pattern formats, such as the *Alexandrian form* [45], *GOF form* [131], or *POSA form* [67], have become widely accepted and used by other authors either unchanged, adapted, or extended [98, 99, 121, 124, 159].

[3] Depending on the author, the pattern format is also referred to as pattern form [45, 159] or pattern template [67, 131].

The *Alexandrian form* follows a narrative style and contains the following sections [10, 45]:

- A picture to show a typical example of the pattern.
- An introduction to set the context for the pattern.
- A headline to explain the problem in one or two sentences.
- A problem body to describe the empirical background of the pattern, the range of ways to manifest the pattern, the evidence for its validity, and a set of forces involved in resolving the problem. Forces can be interpreted as requirements and constraints that the pattern must be balanced within its context [67].
- A solution described in the form of instructions.
- A diagram with labels to illustrate the solution and the main components of the pattern.
- A discussion of how the pattern is connected and related to other patterns in the pattern language.

In addition to the elements, each pattern is associated with a name describing what the pattern accomplishes. Compared to the *Alexandrian form*, the *GOF form* focuses on design patterns that describe *"communicating objects and classes that are customized to solve a general design problem in a particular context"*. By defining a larger number of elements, the pattern format becomes more structured. The sections building the *GOF form* are [131]:

- Pattern name and classification.
- Intent: Statement about the goal and design issue addressed.
- Also Known As: Other names of the design pattern.
- Motivation: A scenario to illustrate the design problem and how it will be solved.
- Applicability: Description of the situation in which the design pattern can be applied.
- Structure: A graphical representation to illustrate the structure of the design pattern.
- Participants: Classes or objects that are part of the design pattern along with their responsibilities.
- Collaborations: Description of participants collaborating to fulfill their responsibilities.
- Consequences: Tradeoffs and results when applying the design pattern.
- Implementation: Description of pitfalls, hints, or techniques.
- Sample Code: Example source code to illustrate the implementation.
- Known Uses: Examples of successfully applied design patterns.
- Related Patterns: Relations between the described pattern and other design patterns.

The *POSA form* is structured similarly to the *GOF form* but uses different section names with similar meanings. The elements of the *POSA form* are: *name*, *also known as*, *example*, *context*, *problem*, *solution*, *structure*, *dynamics addressing runtime aspects*, *implementation*, *example resolved*, *variants*, *known uses*, *consequences*, and *see also*. A more detailed comparison and discussion of different design pattern formats can be found in [21, 67, 68].

Pattern Classifications

Different types of patterns in software development and engineering exist, which vary in their granularity, level of abstraction, scope, and purpose. Some patterns may address domain-specific design issues for a particular programming language, whereas others are domain and programming language-independent. In order to group related patterns, pattern classifications have been introduced, for example, in [67], where patterns are categorized into architectural patterns, design patterns, and idioms as follows:

- Architectural patterns refer to a category with the highest abstraction providing templates for concrete software architectures, e.g., predefined subsystems with rules and guidelines for organizing their relationships.
- Design patterns provide solutions for refining subsystems or components of a software system along with their relationships between them. Typically, design patterns are independent of a particular programming language or paradigm.
- Idioms represent low-level, programming language-specific patterns that deal with the realization of particular design issues. The introduced programming and design techniques are often applicable to a particular programming language, such as C++ [82].

Design patterns are further classified in [131] based on their purpose and scope. A design pattern's purpose can be either creational, structural, or behavioral. Creational design patterns describe problems and solutions related to object creation mechanisms, while structural design patterns consider the composition of classes or objects. Behavioral design patterns provide solutions for the interaction and distribution of responsibilities between classes and objects. Additionally, the scope specifies whether the pattern addresses classes or objects.

2.4.2 Related Work on Power and Energy Aspects of Design Patterns

This section provides an overview of existing work related to the energy-aware design pattern framework presented in Chapter 4 (p. 89 ff.). Existing research considers power- and energy-related aspects of software design patterns from two orthogonal perspectives. The first perspective focuses on *improving* design patterns w.r.t. their impact on power and energy consumption. The second perspective focuses on *enhancing* software applications by domain-specific design patterns addressing power- and energy-related aspects.

Improvement of Design Patterns

Several approaches aim to analyze the effect of structural software design patterns described in [131] on power consumption when applied to software applications. In [224], the power consumption and performance of software applications for embedded systems are analyzed before and after design patterns such as *factory method*, *observer*, and *adapter* have been applied. Another approach by Maleki et al. (2017) [234] compares the power consumption of object-oriented programming concepts, e.g., inheritance, polymorphism, dynamic binding, and overloading, as well as different software design patterns, such as *decorator*, *flyweight*, and *facade*, which are utilizing or based on those concepts. Other publications have also dealt with the impact of various software design patterns defined in [131] on power consumption using C++ [331] and Java [64] as programming languages. For instance, Feitosa et al. (2017)

[122] analyzed the impact of different well-known behavioral design patterns, e.g., *template method* and *strategy*, on energy consumption and implemented respective alternative design solutions for comparison with expected lower power consumption. Approaches such as [263] aim to optimize software design patterns automatically at compile time and, thus, close to the instruction level by optimizing object creations, function calls, and memory accesses while taking compiler optimizations automatically into account.

The analyzes of design patterns presented in [64, 122, 224, 234, 331] focus on implementation-specific issues based on mechanisms of object-oriented languages such as inheritance, polymorphism, and overloading. Since the realization and the effects of such mechanisms may differ for specific programming languages such as C++ and Java, the approaches consider the realization of the design pattern, e.g., as idioms, rather than the concept of the design pattern itself. However, the impact on power consumption can vary widely for different programming languages and concepts. Since the compiler, e.g., settings and optimization levels, also directly affects power consumption, these findings cannot be generalized. This is confirmed in [4], where results showed that the change of programming languages and compiler settings heavily influence the overall performance and the impact of the software on power consumption in particular. Bunse and Höpfner (2008) [63] stated that source code optimization during compile time is often inefficient since the usage of existing resources cannot be predicted. Moreover, accessing and using connected resources, such as peripheral devices, is an important factor of power and energy efficiency considerations for embedded systems, which is not covered by the analyzed design patterns in related work.

Related work discussed in this section focuses on the MCU and low-level aspects of object-oriented programming languages, which are not generalizable due to the variance of programming languages and compiler setting possibilities. The energy-aware design patterns specified with the novel design pattern template as part of the framework presented in Chapter 4 (p. 89 ff.) do not represent language-specific design pattern implementations or idioms but rather common solutions to design issues related to energy and power. Due to this, discussed related work is considered out of the scope of this thesis.

Enhancement by Design Pattern

This section deals with design patterns and pattern templates aiming to enhance a quality aspect of the system, such as power consumption. The main drawback of behavioral software design pattern templates introduced in Section 2.4.1 (p. 32 ff.) is the lack of sections to cover quality aspects used in typical NFRs, e.g., power consumption and time behavior as important aspects of embedded systems.

The general relationship between software design patterns and NFRs is discussed in [145]. The authors propose goal graphs for design patterns as a systematic approach to describe properties and a reasoning structure of design patterns with NFPs as goals. The traces defined in [145] between the non-functional goals of a pattern may be used to evaluate, analyze, and discuss NFP-related design decisions during the system design phase.

In [204], the authors introduce the concept of requirements patterns. Based on a modified variant of the design pattern template introduced in Section 2.4.1 (p. 32 ff.), sections covering constraints have been added, while other sections, such as *implementation* and *sample code*, have been removed from the template. The *constraints* section contains functional or non-functional restrictions that are applied to the system. Examples listed in [204] include hardware, timing, and environmental constraints. The basic concept has also been applied to object

analysis patterns in follow-up work [205], in which the *constraints* section includes specifications of different property types written in a temporal-logic-based format that should be fulfilled when using a given pattern.

In common representations introduced in Section 2.4.1 (p. 32 ff.), only time-related aspects are briefly addressed for a subset of patterns, e.g., in [98]. However, none of the aforementioned approaches considers the impact on NFRs as part of the software design pattern template. The approach in [21] extends common representations to consider safety aspects as part of NFRs in the pattern description for safety-critical applications. It also provides a quantifiable measure for safety-related NFRs, such as reliability, safety, costs, modifiability, and execution time. However, power- and energy-related aspects are not part of the proposed template.

In [307], the authors identify energy-efficient software solutions, which are transferred into a catalog of green architectural tactics in subsequent work [308, 309]. These tactics refer to domains such as energy monitoring, self-adaptation, and cloud federation specifically to address cloud software architectures. The design pattern template includes the fields *motivation*, *description*, *constraints*, *example*, and *dependency* for the textual description.

An analysis and a catalog for synchronization design patterns for multi-threaded software applications are presented in [228]. The main idea is to use DVFS for each design pattern to achieve a more energy-efficient solution. The pattern template includes fields for the *pattern name*, a *figure*, the *main strategy*, and *example scenarios*. However, the proposed design patterns are limited to MCUs with DVFS support.

Reinfurt et al. (2016) [321] propose a pattern framework and catalog describing abstract design patterns for IoT. In later work, the authors also presented general design patterns for IoT devices [322], for powering, operating, and sensing of IoT devices [323], bootstrapping and registration [324], and communication and management in IoT [325]. The catalog includes categories partly based on [51], like device operating modes, energy supply types, e.g., event-based harvesting and battery- or mains-powered, communication, management, and sensing for design patterns that address the characteristics and the behavior of IoT devices. Exemplary IoT design patterns of the catalog *schedule-based sensing*, *normally-sleeping device*, and *device wakeup trigger*. The proposed design pattern template includes fields such as the *name*, *aliases*, *icon* for a graphical representation, *context*, *problem*, *forces* to describe considerations that must be taken into account, *solution*, *variants*, *related patterns*, and an *example implementation* as key aspects of the design pattern. Energy efficiency is considered w.r.t. energy harvesting and energy-saving approaches in system architectures of IoT ecosystems, such as server systems and infrastructure. However, to the best of our knowledge, the aforementioned work does not consider the actual software application of IoT nodes and their close connection to the hardware layer.

Further work related to this thesis is presented in [85, 86]. The authors propose design patterns to develop more energy-efficient software applications for mobile devices. Examples of proposed design patters are *WakeLock: Incorrect wake lock usage*, *UnusedResources*, *Push over Poll*, and *Cache*.

Although energy-aware source code optimization is not in the scope of this thesis, it should be noted that design patterns may also be expressed as a set of design rules for specific hardware components. For instance, the Ultra-Low Power Advisor [378] is a tool released by Texas Instruments to assist software developers in creating more energy-efficient source code for the MSP430 MCU family [379]. The tool provides a set of rules, for example, to reduce excessive cycles of the MCU, to encourage the use of specific functional blocks such as the *Direct Memory Access (DMA)*, and to avoid uninitialized interfaces such as GPIOs. While

some rules are generally applicable and independent of the MCUs architecture, others may be product-line or vendor specific. However, the design rules are limited to MCUs, which are not necessarily the primary source of energy consumption within an embedded system [294].

The approaches discussed in this section, e.g., [85, 86, 228, 323], address the impact of design patterns on the power consumption of the system. However, to the best of our knowledge, related work does not include power consumption and execution time as closely related aspects in the description of software design patterns. Additionally, the approaches lack metrics to specify the impact of design patterns on NFPs, especially power consumption, as the subject of NFRs. To address this gap and answer RQ2, Chapter 4 (p. 89 ff.) proposes a novel software design pattern framework to describe best practices as energy-aware software design patterns in a comprehensive, quantifiable, and comparable manner.

2.5 Model-driven Development (MDD)

This thesis aims to provide a workflow for an early power consumption estimation of software application models in *Model-driven Development (MDD)*. As a methodology for a software development process, MDD is often also referred to as *Model-driven Software Development (MDSD)* [369]. However, since the term MDD is more common than MDSD, MDD will be used in the remainder of this thesis. It describes a software development methodology that uses models as the primary artifact of the development process and not just for documentation purposes [56]. The increasing complexity of software applications makes the development process error-prone and costly [282]. Compared to traditional code-centric approaches, MDD provides advantages such as increased development speed and productivity [193] and enhanced software quality due to the partial or complete generation of source code. Furthermore, MDD may improve the manageability of complexity through abstraction [369]. The code generation process is automated in theory, but in practice, it is often semi-automated with manual post-processing. MDD is related to other model-driven concepts, as shown in Figure 2.7.

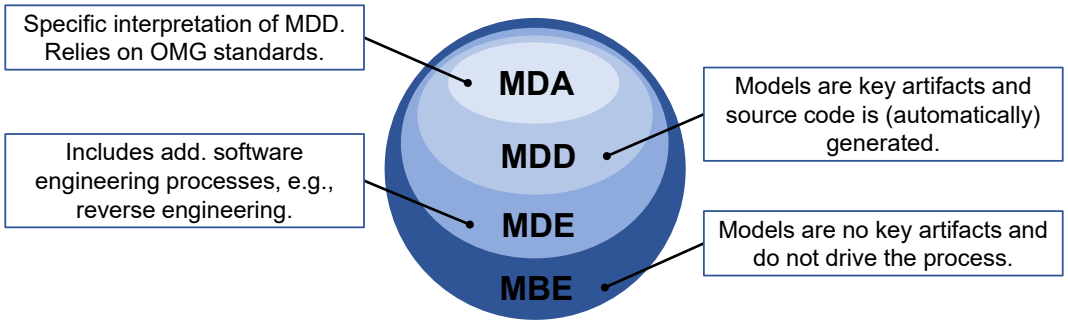


Figure 2.7: Relations between the model-driven concepts, adapted from [56].

In *Model-based Engineering (MBE)*, also denoted as *Model-based Development (MBD)*, models are not the key artifacts and are used, e.g., in the analysis phase, to define the system while they serve as templates for manual code-writing activities [56]. *Model-driven Engineering (MDE)* is a superset of MDD and includes other aspects of the engineering process, such as model-driven reverse engineering [56]. A concept closely related to MDD is *Model-driven Architecture (MDA)*. Since MDA relies on standards for modeling and transformation

languages introduced by the *Object Management Group (OMG)*, such as the *Meta Object Facility (MOF)* (cf. Appendix A.1.2, p. 264 ff.) and the UML (cf. Appendix A.3, p. 266 ff.), it can be considered as a subset of MDD and a particular vision of the OMG [272]. However, the terminology for MDA introduced by the OMG is suitable to further explain the basic concepts of MDD.

Section 2.5.1 discusses modeling languages used in MDD, while Section 2.5.2 summarizes the concept of model transformations. Additional background on the architectural layer and the abstraction levels of MDD, including the definition of *Computing Independent Model (CIM)*, *Platform Independent Model (PIM)*, and *Platform Specific Model (PSM)* and the relation between models, metamodels, and meta-metamodels, can be found in Appendix A.1 (p. 263 ff.).

2.5.1 Modeling Languages

This section discusses modeling languages in MDD, focusing on characteristics such as extensibility, domain independence, and popularity. Extensibility targets the property of a modeling language to provide mechanisms for extending the modeling language itself or specific concepts of the language. The extensibility of a modeling language is especially important for addressing RQ3 (cf. Section 1.2.2, p. 8). Modeling languages can be domain-dependent, with concepts only applicable within the specific domain. A more general intention of this thesis is domain independence and, thus, the ability to adapt the presented concepts to different domains. Because of this, a general-purpose modeling language is preferred, which also impacts RQ3 and RQ4. The popularity, however, indicates a modeling language's suitability in academic and industrial fields. While being a more tenuous criterion, the popularity of a modeling language is correlated to its tool support. A comparison of MDD tools for UML and their modeling languages support is provided in Appendix A.2 (p. 24 ff.) since tool support is not directly related to any characteristic of the modeling language itself. In the following, the criteria and characteristics mentioned above are discussed for a selection of available modeling languages.

The ***Unified Modeling Language (UML)*** [275] is a general-purpose modeling language standardized by OMG used to specify, design, implement, and analyze software-based systems. The UML specification provides a standardized semi-formal metamodel for modeling object-oriented systems and a set of human-readable diagram types as a graphical notation to visualize elements and aspects of such models. With a focus on software engineering and software-based systems, the UML diagram types may be applied in different phases of the engineering process to describe structural and behavioral aspects of a software-based system. Moreover, UML is independent of any specific domain, programming language, target platform, or development tool by definition. With profiles, however, UML provides a generic mechanism for extensions, which may be used to refine the standard notation of the UML metamodel and, e.g., to add domain-specific concepts and notations. For this, profiles contain stereotypes, tagged value definitions, and constraints that can be applied to various UML elements. UML has been identified as the de-facto standard language for modeling software-based systems [74, 213] and also as the most widely used modeling language in the embedded software industry [7, 407].

The ***Foundational UML (fUML)*** [281] is another standard specified by the OMG and specifies foundational execution semantics. As a subset of the UML, fUML contains parts of the UML packages for composite structures and classes to model the structure of a system as well as activities, common behavior, and actions for the behavior of the system. Along with the *Action Language for Foundational UML (ALF)* [277] to specify executable

behaviors using primitive types, actions, and control flow mechanisms, fUML provides precise semantics for executing the UML-based model within a virtual machine. Profiles and, therefore, stereotypes are stated as out of scope in the specification [281], which limits the extensibility of fUML to model domain-specific and non-functional aspects. The fUML specification does not cover UML state machines, but a definition of the execution semantics is available through extensions [280]. The lack of support for profiles and stereotypes has been addressed by researchers, i.e., in [44, 377]. However, the OMG has not considered the proposed solutions for inclusion in the specification, nor have they been adopted by vendors or developers of MDD tools.

The ***Systems Modeling Language (SysML)*** [279] is a general-purpose modeling language for systems engineering applications adopted and further refined by the OMG. Utilizing the profile mechanism of UML, SysML is itself defined as a profile and therefore adopts parts of the specification and semantics of UML. SysML reuses some UML diagrams (e.g., state machine diagram and package diagram) and introduces modified diagram types, such as the block definition diagram, which is based on the UML class diagram. While UML focuses on software engineering, the scope of SysML is systems engineering, which includes areas such as requirements engineering and system modeling, e.g., hardware, mechanical, and software parts. For this, SysML introduces two new diagrams: the requirements diagram and the parametric diagram. According to Akdur et al. (2018) [7], SysML is less common in the embedded software industry than traditional UML and UML profiles such as MARTE.

The ***Modeling and Analysis of Real-Time and Embedded systems (MARTE)*** [278] UML profile is an extension of UML and a *Domain-specific Language (DSL)* specified by the OMG intended to support the modeling and analysis of real-time and embedded software applications in MDD. UML has a software-centric view from a logical perspective while lacking the ability to describe the properties and characteristics of the underlying embedded systems [348]. MARTE fills this gap and provides extensions, e.g., for real-time modeling and performance and scheduling analysis of system designs in a standardized format. More interestingly, MARTE introduces the specification and modeling of NFPs based on energy consumption, provides time modeling aspects, and enables a high-level analysis, e.g., of software-related behavior on aspects such as timing, power, and energy. The specification offers a set of predefined NFP definitions while the definition of custom NFPs is also supported. MARTE can be applied to both UML and SysML. Further aspects of MARTE related to the concepts presented in this thesis are discussed in-depth in Section 2.6 (p. 41 ff.).

Simulink & Stateflow are parts of the MATLAB platform developed by MathWorks [380]. Simulink [383] is a graphical modeling language for the model-based development of dynamic systems. Models in Simulink are defined as hierarchical block diagrams and can be simulated by a simulation engine. The main concepts of Simulink are signal flows for time-continuous and time-discrete systems, whose models consist of, e.g., differential equations and difference equations, respectively. The simulation is based on so-called solvers to compute the states of a model. Stateflow [384] is another modeling language that extends Simulink and provides event modeling for discrete systems. To achieve this, Stateflow introduces state transition diagrams and flow charts. Simulink and Stateflow are proprietary modeling languages with graphical programming environments based on and provided by MATLAB [380]. Both languages are not restricted to a specific domain. Since they are proprietary and part of MATLAB, the extensibility of both languages is limited. However, there are attempts in research [149, 210, 376] to achieve a model transformation (cf. Section 2.5.2, p. 40 ff.) between UML and Simulink/Stateflow models.

2.5.2 Model Transformations

Besides the definitions of models, model transformations are also an important aspect of MDD and MDA, which allow the definition of mappings between different models, e.g., PIM to PIM or PIM to one or more PSMs. In general, MDA distinguishes between model-to-model and model-to-text transformations. The following formal description of model transformations is based on a unified notation originally published in [69, 70] and [229].

Let s be a system, m a model, and fm a formalism so that $m(s)/fm$ defines a model of a system using the specific formalism fm. To describe a formalism's model, a metamodel may be defined as $mm_1(fm_1)/fm_2$ to denote a metamodel mm_1 of the formalism fm_1 expressed in the formalism fm_2. For instance, the metamodel mm(UML)/MOF uses the MOF to define and describe the meaning of any model $m(s)/UML$. For more information about the architectural layer of MDD and metamodels, see Appendix A.1 (p. 263 ff.).

Model transformations allow the definition of mappings between different models. In general, transformations are defined at the metamodel level and applied on the model level, which enables the reuse of the transformation for all models that conform to the metamodel addressed by the transformation. A mapping between m_1/fm_1 and m_2/fm_2 is expressed as $m_1/fm_1 \rightarrow m_2/fm_2$ and requires a set of mapping rules $R_{mm_1 \rightarrow mm_2}/fm_R$, which are also described using a formalism fm_R. Note that transformations may be considered as models themselves and can be expressed as $m(t)/fm_t$ with t as an instance of a transformation and fm_t as the formalism, e.g., the *Atlas Transformation Language (ATL)*. Due to this, the execution of a transformation between two models may be defined as a function $t(m_1/fm_1, R_{mm_1 \rightarrow mm_2}/fm_R) \rightarrow m_2/fm_2$, where a transformation instance executes the transformation process based on the mapping rules. The concept of model transformation is shown in Figure 2.8. According to [246], model transformations may be defined as a multi-dimensional process, where the most common dimensions describe:

- The number of source and target models involved.
- The type of metamodel used, denoted as endogenous and exogenous transformation.
- The abstraction level, described as horizontal and vertical transformation.

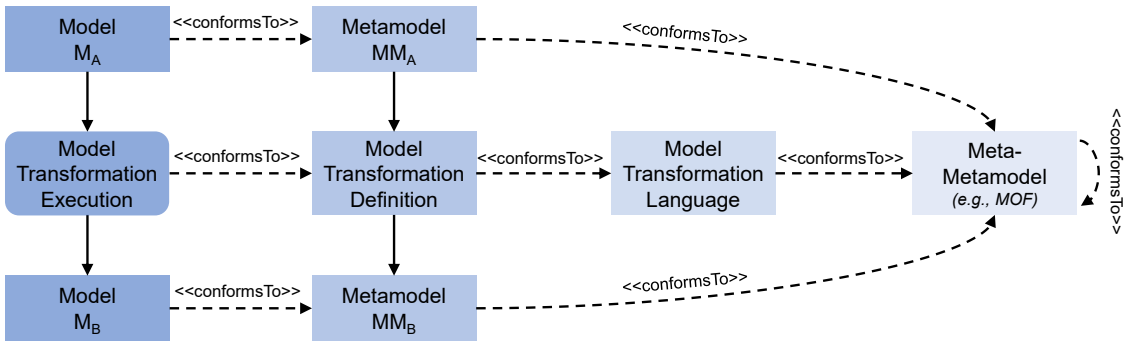


Figure 2.8: Transformation process between models, adapted from [56]. Artifacts are illustrated as rectangles and the model transformation activity is pictures as rectangle with rounded corners. Solid lines describe the transformation process while dotted lines indicate the conformance between artifacts.

In model-to-model transformations, models are used as the input and output of the transformation process. The transformation may be performed on multiple target and destination models, defined as one-to-one, many-to-one, or one-to-many transformation. Furthermore, model transformations can be endogenous or exogenous. Imagine two models, m_i and m_j, confirming to metamodel mm_i and mm_j, respectively. In endogenous transformations, the metamodel of the input model is equal to the metamodel of the output model, i.e., $m_i = m_j$, while the metamodel differs ($m_i \neq m_j$) for exogenous model transformations. In fact, exogenous model transformations require a set of rules which describe how elements of the input model are mapped to the elements of the output model. If a transformation rule does not address elements of the input model, they will be ignored during the transformation process, and no element will be created in the output model. The last dimension mentioned above distinguishes between horizontal and vertical transformations. In horizontal transformations, the abstraction level between the input model m_i and m_j remains the same, e.g., when refactoring or migrating models. Vertical transformation refers to a transformation where the input and output models have a different level of abstraction. For instance, transforming one PIM into multiple PSMs is described as a vertical exogenous one-to-many transformation, e.g., $m/\text{UML} \rightarrow (m/\text{C++}, m/\text{Java})$. A model exchange between different modeling tools is an example of a horizontal transformation. As described in [179, 180, 181, 182], the analysis of models, e.g., developed in UML or Simulink, may also require model-to-model transformations. Code generation, for instance, may be described as a vertical transition from models, e.g., a PSM, to lower-level artifacts, e.g., C++ or Java source code, which is a typical example of a model-to-text transformation [56].

2.6 Modeling of Embedded Systems with UML and MARTE

To apply MDD in software engineering, modeling languages are used to specify, visualize, and document models. As a general-purpose graphical modeling language, UML has been established as the standard modeling language in MDD and represents the most used modeling language in the embedded software industry [7, 407]. As a central part of this thesis, UML is used to model hardware components (cf. Section 5.2, p. 117 ff.), specify the UML-based *Power Analysis Profile (PAP)* (cf. Section 5.3, p. 124 ff.), and define the software application model of the case study (cf. Section 7.2, p. 176 ff.). A description of the UML diagrams used in this thesis can be found in Appendix A.3 (p. 266).

This section discusses the MARTE UML profile as the basis for the definition of the PAP. A brief overview of MARTE is given in Section 2.6.1. The basic structure and profiles of MARTE are introduced in Section 2.6.2, while Section 2.6.3 covers the *Value Specification Language (VSL)*, a textual language to specify quantitative information. The modeling of NFPs is discussed in Section 2.6.4.

2.6.1 Overview

UML can be customized and extended with the concept of UML profiles (cf. Section A.3.2, p. 270 ff.) to take domain-specific aspects into account. A UML profile generally represents a domain-specific interpretation of the general UML language. While focusing on functional aspects, basic UML offers no support to model NFPs nor NFRs in software development. Two early UML extensions standardized by the OMG which are using NFR-specific annotations [46, 108] for the analysis of NFPs are the *UML Profile for Modeling Quality of Service*

and Fault Tolerance Characteristics and Mechanisms (QoS&FT) [271] and the *UML Profile for Schedulability, Performance, and Time Specification (SPTP)* [270].

The SPTP profile extends UML with basic *Quality of Service (QoS)*[4], concurrency, resource, and timing concepts specific to real-time systems. Furthermore, SPTP adds requirements and properties needed for the two types of quantitative analysis, namely, schedulability and performance [46]. The QoS&FT profile has a broader scope than SPTP, allowing a larger extent of QoS requirements and properties to be considered, which may be fixed at design time or managed dynamically [46, 91]. However, the QoS&FT profile neither includes typical results of scheduling analysis like end-to-end response time and the schedulability of tasks when describing workloads [271] nor enables tools to extract basic concepts like triggers or tasks [109] needed to fully support the process of analysis modeling. Multiple sources [108, 110] have referred to the two-step annotation process as being too heavy-weight and requiring too much effort compared to SPTP. Furthermore, the profile lacks variable specifications. The additional objects to be created for annotation purposes and the use of long *Object Constraint Language (OCL)* expressions may also negatively affect the readability of the model[5].

The OMG issued a request for proposal to upgrade the SPTP to the UML 2.0 specification [108]. As a result, the MARTE specification has been released as a direct replacement for the SPTP [108, 278, 348]. MARTE is based on features of the SPTP and reuses structural components of the QoS&FT and corresponding stereotypes of the SysML profile. As stated in [348], MARTE enhances UML with several capabilities, such as:

- Methods to define and specify types of quantitative and qualitative NFPs and their relationships for UML models and model elements.
- A generic framework to model NFPs, including power-related properties.
- The ability to model hardware resources accurately.
- Methods to model software applications specifically for real-time and embedded systems.
- Methods to model the relationship between the software application and the underlying hardware platform.

With the capabilities added by MARTE, the basis for a formal analysis may be defined, which enables the ability to predict and evaluate key aspects such as power consumption automatically or semi-automatically in early design phases before a hardware platform may be available, which addresses RQ3 and RQ4. In the following, the basic structure of MARTE will be introduced, and aspects such as the *Value Specification Language (VSL)* and the modeling of NFPs will be explained in detail.

2.6.2 Basic Structure and Profiles

According to [278], the definition of profiles in MARTE relies on a two-stage process, namely the domain model specification and the definition of underlying UML profile design:

- The first stage covers the definition of domain elements representing required concepts related to a specific concern, e.g., NFP modeling. A metamodel and detailed semantics descriptions of each containing element formalize the resulting domain model.

[4] In this context, QoS is equivalent to NFPs.

[5] A more detailed comparison of the SPTP and the QoS&FT profiles may be found in [46, 108, 109].

- In the second stage, UML profiles are designed as UML-based representations of MARTE domain model elements.

Profiles defined by MARTE provide extensions based on stereotypes, tagged values, and specific notations intended to integrate a complementary DSL for designing, analyzing, and building embedded and real-time systems into the concepts of UML. The general structure of MARTE is shown in Figure 2.9. Packages and profiles (partly) used in this thesis are highlighted in blue.

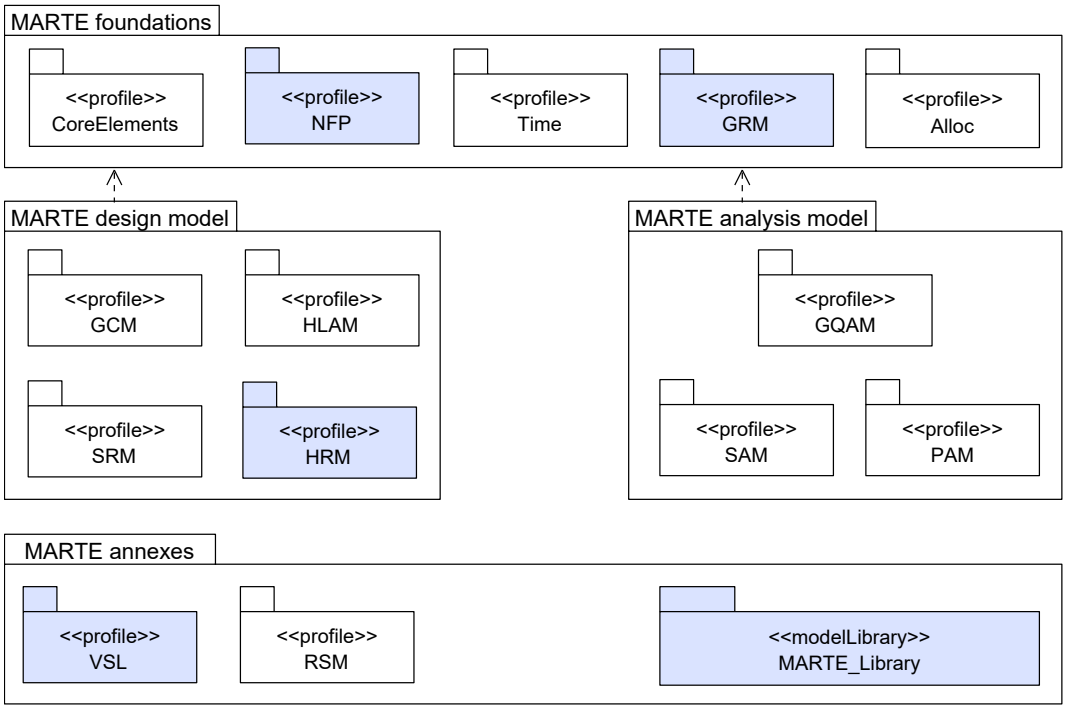


Figure 2.9: MARTE architecture and profile overview, adapted from [278]. Elements (partly) used in this thesis are highlighted in blue (UML 2.5 package diagram notation).

To address RQ3 and RQ4, MARTE is used for two basic concepts: the definition of the PAP and the specification of NFPs for hardware components (cf. Section 5.3, p. 124 ff.). For the PAP, stereotypes of different MARTE profiles, such as the *Generic Resource Model (GRM)* and the *Hardware Resource Modeling (HRM)*, are considered, which provide tagged values for the definition of NFPs.

The GRM profile is a core framework of MARTE used to model a general platform for executing embedded applications. For instance, the MCU can be a resource that provides processing capabilities as a resource service. The use or demand of resources is addressed by the `ResourceUsage` package of the GRM, which enables an evaluation of requirements based on resources needed during execution. For this, the metamodel of UML can be extended with stereotypes of the GRM. However, the utilization of the GRM profile in the context of this thesis is limited to the `Resource` and `ResourceUsage` stereotypes, as shown in Figure 2.10.

Besides the general `Resource` stereotype, the GRM package also provides specializations such as `StorageResource`, `TimingResource`, `ComputingResource`, and `DeviceResource` [278]. The `ResourceUsage` stereotype contains tagged values that may be suitable

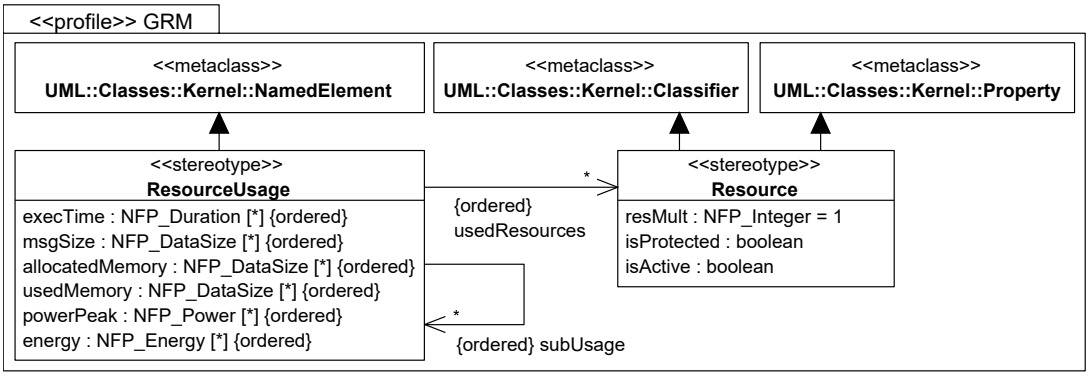


Figure 2.10: Excerpt of the `Resource` and `ResourceUsage` stereotype definitions (GRM profile), adapted from [278] (UML 2.5 profile diagram notation).

to model aspects related to power and timing. Furthermore, the stereotype may be a reasonable basis for more complex scenarios in which multiple resources of different types are used. Such information can be modeled within a single stereotype instead of being spread across the model [348]. Since the `ResourceUsage` stereotype is an extension of the UML `NamedElement`, it can be applied to nearly all elements of the UML model, including classes, state machines, states, and transitions.

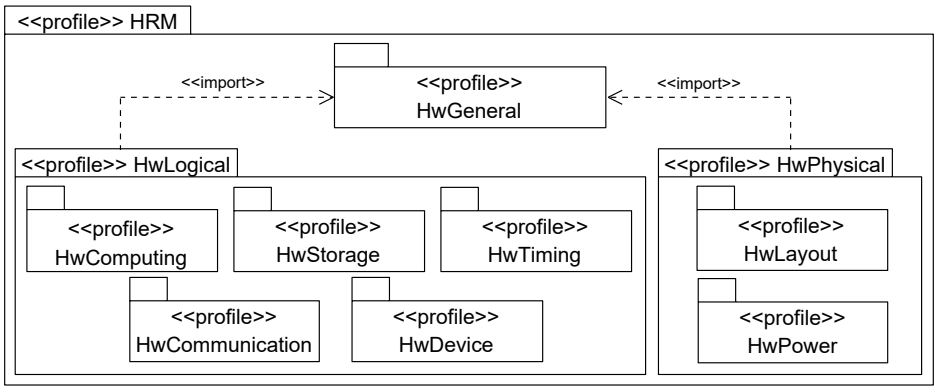


Figure 2.11: Structure of the HRM profile, adapted from [278] (UML 2.5 package diagram notation).

A specialization of the GRM profile is the HRM profile to model hardware parts of embedded system platforms. The HRM profile illustrated in Figure 2.11 aims to structure hardware concepts under a hierarchical taxonomy with categories depending on their structure, functionality, and technology [278]. For this, the HRM offers two views: a logical view (`HwLogical`) to classify hardware resources by their functional properties and a physical view (`HwPhysical`) for a classification related to physical properties. The HRM profile illustrated in Figure 2.11 consists of the following packages (sub-profiles):

- The `HwGeneral` package provides a `HwResource` stereotype, which inherits from the `Resource` stereotype (GRM package) and provides at least a `HwResourceService`, as

illustrated in Figure 2.12. The stereotypes `HwComponent` and `HwResourceService` provide tagged values to describe NFPs such as power consumption and heat dissipation.

- The `HwLogical` package consists of sub-profiles such as `HwComputing` to model, e.g., MCUs, `HwStorage` to model, e.g., memory, and `HwCommunication` for communication interfaces. Further sub-profiles of the package are `HwTiming` for clocks and timers and `HwDevice` to model sensors and actuators.
- The `HwPhysical` package consists of the sub-profiles `HwLayout` and `HwPower`. The `HwLayout` sub-profile provides a generic `HwComponent` stereotype to describe physical components with tagged values such as `dimension`, `position`, `price`, and `weight`. With the `HwPower` profile, models may be annotated with power properties.

Figure 2.12 illustrates the `HwPower` sub-profile as part of the `HwPhysical` profile. The tagged values `staticConsumption` and `staticDissipation` define the consumption and dissipation of a non-operating or non-active hardware component, while the tagged values `consumption` and `dissipation` define the (dynamic) consumption and dissipation of the `HwComponent` when powering the `HwResourceService`. Furthermore, the `HwPower` sub-profile shown in Figure 2.12 defines the `HwPowerSupply` stereotype, which inherits from the `HwComponent` to model aspects of power supplies and batteries. Note that the layout and physical-specific attributes such as `position`, `weight`, and `price` of the `HwComponent` stereotype have been omitted in Figure 2.12. For the description of NFPs, the VSL and NFP profiles and the MARTE model library are used (cf. Figure 2.9, p. 43). Aspects of the VSL and NFP modeling are discussed in the following.

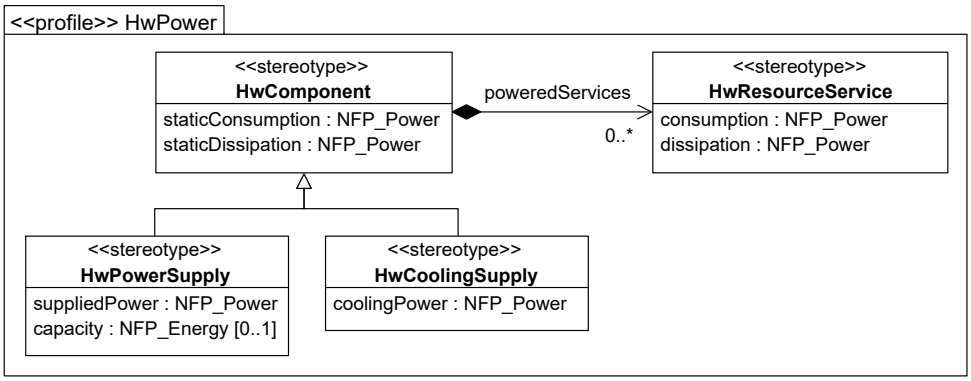


Figure 2.12: The `HwPower` sub-profile as part of the HRM profile, adapted from [278].

2.6.3 Value Specification Language (VSL)

MARTE introduces the *Value Specification Language (VSL)* to specify non-functional aspects based on quantitative information. The VSL is used to specify values for properties of stereotypes and to define constraints and complex data types. Since the *Tag Value Language (TVL)* of the SPTP lacks the ability to model user-defined NFPs, the VSL extends concepts of the TVL, e.g., to annotate constant, variable, tuple, and expression values [278]. According to the OMG specification for MARTE [278], VSL provides concepts to specify parameters, constants, and expressions in a textual form. Additionally, the VSL is able to express relationships

between elements by using arithmetic, logical, relational, and conditional expressions. To extend UML with complex data structures, the VSL provides stereotypes for composite data types such as collection, interval, choice, and tuple types. By this, values can be specified with the VSL as *literal data types* representing single values and *composite data types* for structured data types with additional attributes.

- Literal values represent basic data types in text-based formats that are common to most programming languages, such as boolean, string, and number (integer, real) literals.
- The VSL predefines four composite data types. The `IntervalType` data type defines a valid range of data values with two boundary values or as an enumeration. The `CollectionType` defines data structures as arrays, the `ChoiceType` combines structurally unrelated data types into a single type, and the `TupleType` expresses ordered collections of typed attributes [348]. A formal specification of each type can be found in [278].

VSL is also used to define variables and describe relationships between attribute values. According to the MARTE specification [278], VSL variables must be explicitly declared using a dollar sign ($) as a keyword for variable declaration, placed before the variable name. Optionally, a direction information can be defined before the variable name using the keyword `in` for an input variable, `out` for an output variable, or `inout` if the variable is used for both. Also optional are the data types, which can be specified with a colon after the variable name, and the init expression, which can be specified with an equal sign. Listing 2.1 shows the definitions of three variables using a literal data type (line 1), a composite data type with and without an init expression (lines 2 and 3), and an expression assignment (line 5). The `TupleType` defines the basic structure for defining NFPs, e.g., used for the variables `currentTime` and `threshold` in Listing 2.1, and is explained in the following Section 2.6.4.

```
1    out $setAlarm : Boolean
2    in $currentTime : NFP_Duration
3    $threshold : NFP_Duration = (500, ms)
4
5    setAlarm = (currentTime > threshold) ? true : false
```

Listing 2.1: Examples of VSL variable definitions and expression usage.

2.6.4 Non-functional Properties

For the modeling of NFPs, MARTE provides a domain model and a UML profile. The UML profile diagram in Figure 2.13 shows an excerpt of the NFP profile used in this thesis, namely the stereotypes «`unit`», «`dimension`», and «`nfpType`». The «`unit`» stereotype can be considered as an extended description of the entity to be modeled. Properties introduced by the «`unit`» stereotype are [348]:

- `baseUnit` *(optional)*: Symbol representing the base for the measurement unit to be modeled, e.g., W or J as defined in Section 2.1.1 (p. 17 ff.)
- `offsetFactor` *(optional)*: Numerical offset for the conversion between the units of the measurement unit, e.g., between Celsius and Fahrenheit [348].

- `convFactor` *(optional)*: Quantitative relationship between the `baseUnit` and other units of this the measurement unit, e.g., 10^{-3} for the conversion from mW to W.

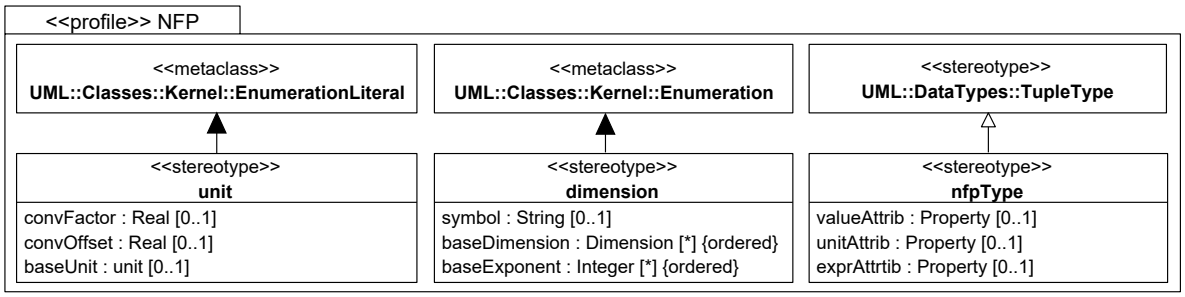


Figure 2.13: Excerpt of the MARTE NFP profile, adapted from [278] (UML 2.5 profile diagram notation).

The «`dimension`» stereotype shown in Figure 2.13 extends the UML enumeration concept and represents the composition of a NFP in terms of the three fundamental physical units length (L), mass (M), and time (T), as well as data (D) to measure information. Note that the dimension may be a single fundamental dimension, such as length, or a combination of multiple fundamental dimensions. To model physical information, the «`dimension`» stereotypes provide a set of properties [348]:

- `symbol` *(optional)*: Used for physical base dimensions L, M, T, or D.
- `baseDimension` *(optional)*: An ordered list used for derived dimensions to specify the relationship between the dimension to be modeled and the physical base dimensions, e.g. $\{L, M, T\}$ for the `PowerUnitKind` (cf. Figure 2.14).
- baseExponent *(optional)*: An ordered list used to specify the exponents for each physical base dimension defined in `baseDimension`, e.g., $\{2, 1, -3\}$ for the `PowerUnitKind`.

The «`nfpType`» stereotype as a UML representation of the *NFP_Type* domain element [278] inherits from the VSL `TupleType`. It is used as a detailed specification of quantitative values, e.g., to model physical values. Properties of the «`nfpType`» stereotype include [348]:

- `valueAttrib`: Placeholder to specify a value of NFPs as numerical quantity or string.
- `unitAttrib`: Declaration of the physical measurement unit corresponding to the value specifications of the NFPs, e.g., `PowerUnitKind`.
- `exprAttrib` *(optional)*: Value of NFPs as VSL expression.

With the `MARTE_Library` (cf. Figure 2.9, p. 43), MARTE also provides a standard library, which contains basic NFP data types with appropriate measurement unit definitions to represent physical aspects. Figure 2.14 shows selected elements of the MARTE model library for measurement units and basic NFP types. The MARTE model library can be extended with additional domain-specific and application-specific physical data types and basic physical measurement units. The extension of the MARTE model library is a sub-step for the definition of the PAP discussed in Section 5.3 (p. 124 ff.). The abstract data type `NFP_CommonType`

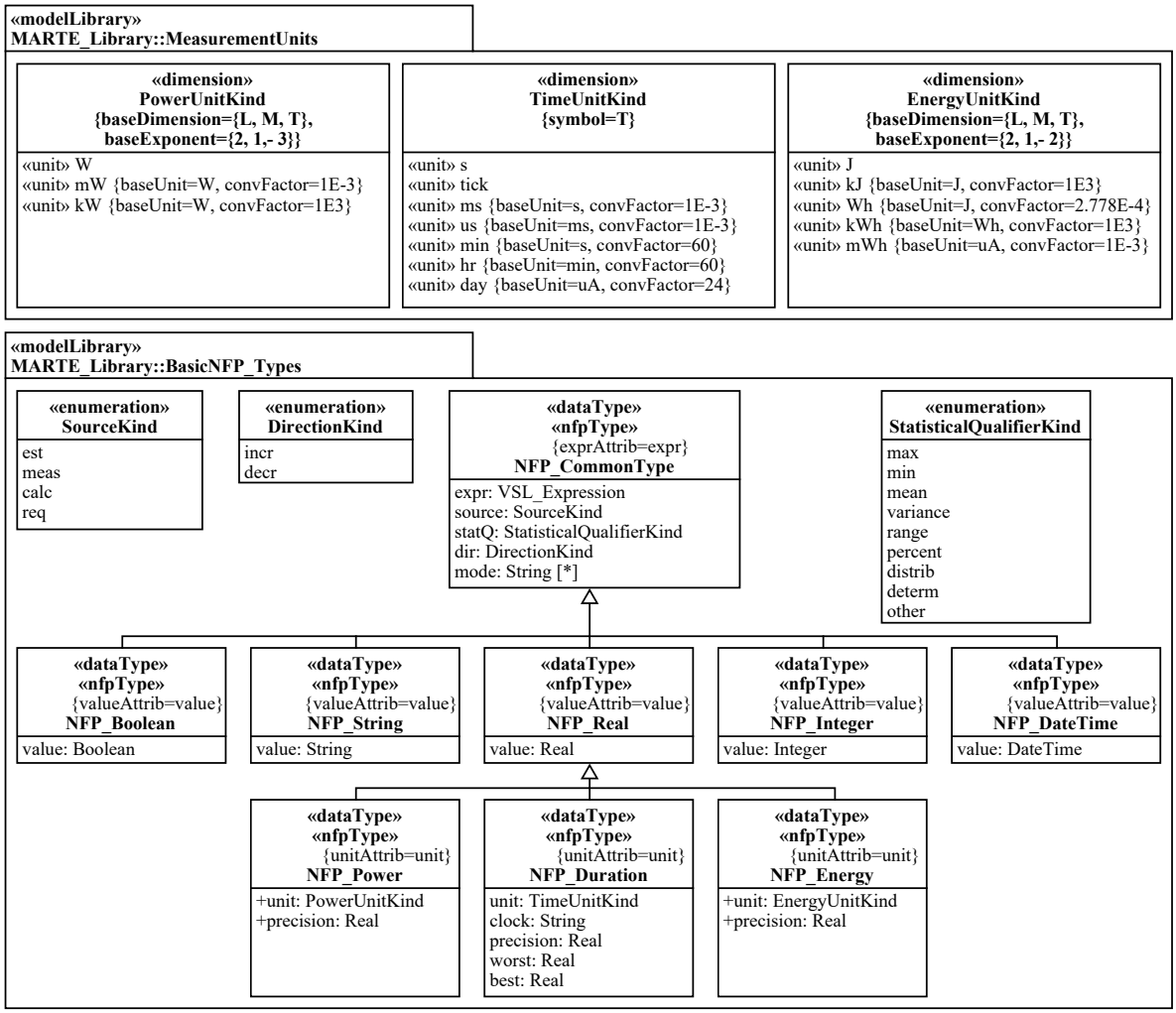


Figure 2.14: Excerpt of the MARTE model library with measurement units and basic NFP data types, adapted from [278] (UML 2.5 package diagram notation).

is defined in the MARTE model library to introduce additional properties as a basis for all derived NFP data types in Figure 2.14, such as `NFP_Power` and `NFP_Energy`. If the additional properties are taken into account, a NFP data type for the MARTE model library may be defined by the VSL `TupleType` $(value, expr, unit, statQ, dir, source, precision)$, with:

- `value`: The actual value expressed as a numerical quantity or string. The `value` property is bound to the default property `valueAttrib` provided by the «`nfpType`» stereotype.

- `expr` *(optional)*: Contains a VSL expression and is used instead of a the `value` property. The `expr` property is bound to the default property `expAttrib` provided by the «`nfpType`» stereotype.

- `unit`: Contains the physical measurement unit. The `unit` property is bound to the default property `unitAttrib` provided by the «`nfpType`» stereotype.

- `statQ` *(optional)*: Statistical qualifier used if the value is a statistical quantity to indicate the type of statistical measure, e.g., max, min, mean, variance, or range.
- `dir` *(optional)*: Enables a comparison for two values of the same NFP type by using the two directions of increasing (`incr`) and decreasing (`decr`) for the order of quality.
- `source` *(optional)*: Describes the origin of the value, e.g., measured (`meas`), estimated (`est`), or calculated (`calc`).
- `precision` *(optional)*: Defines the standard deviation of the real measurement.

As a basic example for the tuple notation, the data type `NFP_Power` may be expressed as $(value = 1.5, expr = -, unit = ms, statQ = -, dir = -, source = -, precision = -)$ or $(value = 1.5, unit = ms)$. If the shortened notation[6] is used, it may be expressed as $(1.5, ms)$.

This section introduced the structure and concepts of MARTE. While UML is used for the functional modeling of software, MARTE extends UML with the ability to model NFPs, which are used in this thesis to describe energy-related properties for power consumption estimation as a form of testing.

2.7 Software Testing Principles

Software testing is an important step in the software development process, which takes up to 50 % of the total time and more than 50 % of the total cost [38, 258]. The approach to estimating power consumption presented in Chapter 5 (p. 115) describes a specific software test method for NFRs partly based on the concepts presented in this section. In addition to a general introduction to the field of software testing, various concepts and methodologies are presented in order to classify the concepts of this thesis and to point out differences with existing approaches.

Since no single definition of software testing exists, the term is interpreted differently in the literature. For instance, Myers et al. (2012) [258] define software testing as *"[...] the process of executing a program with the intent of finding errors"*, while McGregor and Sykes (2001) [241] define software testing as *"the process of uncovering evidence of defects in software systems"*. In standardized and accepted glossaries, (software) testing is referred to as *"activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component"* [174] and as the process of *"[...] all lifecycle activities, both static and dynamic, concerned with planning, preparation, and evaluation of a component or system and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects"* [172]. While focusing on NFRs, aspects of the hardware platform must also be considered in addition to the software application. Since this widely corresponds to the system concept, software testing in this thesis is referred to the definition provided in [174]. A main part of software engineering is verification and validation [235, 367], also known as *V&V* [174]. Verification is located at different steps of the development phase. Based on static methods, verification describes a process to determine whether a unit, component, or system conforms to requirements and standards defined at the specific development phase. Validation is located at the end of the development process and uses software testing methods to check whether a

[6] For readability, the shortened notation $(value, unit)$ and $(expr, unit)$ are used for the rest of this thesis.

component or system solves the assigned problem, meets requirements, and satisfies specified customer expectations. As part of the *Quality Assurance (QA)* process [57], however, software testing is a key element of the entire development cycle used in different phases during the software development process and performed on different abstraction levels. Figure 2.15 shows the most common levels of software tests.

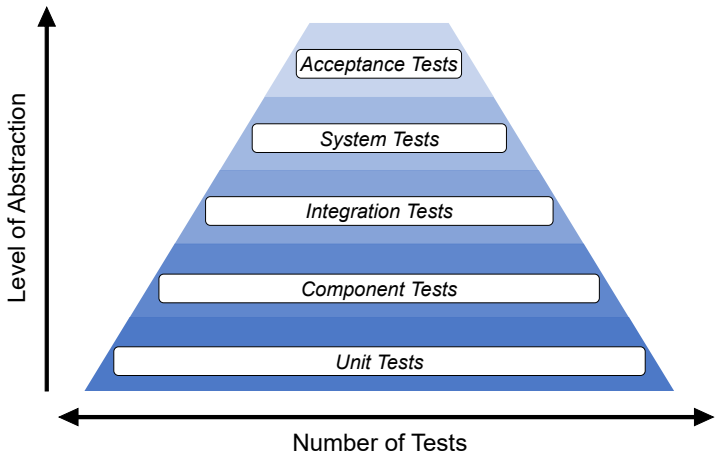


Figure 2.15: Levels of software testing.

Unit testing, for example, is performed on the lowest level to evaluate the functionality of single units as the smallest possible artifacts of a software application against its specification. A single procedure, individual functions in procedural languages, and combinations of operations, e.g., organized in classes as used by object-oriented languages, are typically defined as units [57, 81, 189]. During the test execution, each unit is tested independently and in isolation [235]. Writing tests for all units of a software application results in a large number of test cases. Component testing, or module testing, is similar to unit testing. Instead of individual units, a component as a set of units, e.g., a group of related classes combined in a file or package, is tested independently and in isolation to verify the interaction of contained units and related data structures [16, 394, 397]. Since each unit of a component has already been tested with unit tests, fewer test cases have to be defined. As the next testing level, the integration test focuses on interfaces and interactions between integrated components. In comparison, system testing describes the first stage of testing, where a fully functional SUT, containing the complete software application, hardware components, and documentation, is used to validate the compliance of high-level functional requirements and NFRs [31, 57, 81]. Typically performed by QA teams, system testing focuses on detecting errors at the highest integration level. Besides the architectural design and behavior of the SUT covered by functional requirements, NFRs such as performance, usability, security, reliability, and availability are evaluated [16, 81]. Since no inner details need to be tested, even fewer test cases are required. In acceptance tests, as the highest level of software testing, the software application is evaluated for compliance with the business requirements and tested by end users for its acceptance. Further information about software testing levels can be found in [81, 172, 235, 332, 367].

Section 2.7.1 introduces the dimensions of software testing, while Section 2.7.2 discusses the concept of dynamic testing. Section 2.7.3 introduces *Model-based Testing (MBT)* and describes how this thesis relates to and fits within this systematic testing method. In Section 2.7.4,

X-in-the-Loop (XiL) concepts are discussed, which are used in MDD to evaluate software models. Background on performance analysis and runtime monitoring as an approach complementing conventional software testing is presented in Section 2.7.5. Section 2.7.6 discusses the integration of virtual and physical embedded systems into the testing process.

2.7.1 Dimensions

The characteristics of a software test for a SUT may differ in multiple aspects. A classification based on the granularity level, goal, or technique may be elaborated to describe software tests based on typical characteristics. In previous works [260, 397, 421], domain-specific classification approaches have been presented. For instance, Neukirchen (2004) [260] provides a three-dimensional classification including test goal, distribution, and scope. Software testing describes a vast field, and it is almost impossible to accurately transfer all attributes and aspects of software testing into (orthogonal) dimensions to define a single classification scheme. Nevertheless, Figure 2.16 shows a six-dimensional characterization of software testing as a simplified and more comprehensive representation, elaborated to provide a basic vocabulary for the software testing concepts described in the subsequent sections. The characteristics presented in Figure 2.16 are divided into dimensions, categories, and options, highlighted in bold if used or addressed in this thesis.

The first dimension is referred to as the ***Objective*** [260, 421] and describes the purpose of the test, which can be divided into dynamic and static testing.

> ***Static testing*** includes all testing activities performed on software development artifacts, such as the architecture and source code of the SUT and work created around the SUT, e.g., documentation, requirements, and models [31, 421]. As the term *static* implies, software development artifacts are not executed during testing. Instead, they are statistically analyzed to evaluate logical aspects, such as unreachable code, uncalled functions, or undeclared variables.
>
> ***Dynamic testing*** describes all types of testing where the SUT is executed using test cases. The main goal of dynamic testing is to evaluate the structural aspects and functional and non-functional behavior of a SUT by comparing the result of test cases with the expected results and requirements [260]. Types of dynamic testing include structural, functional, and non-functional testing.
>
> - ***Structural testing*** covers the evaluation of the internal structure of the SUT, e.g., the data flow, during execution. However, the structure must be known in advance in order to derive test cases.
> - ***Functional testing*** aims to evaluate the behavior of the SUT based on functional requirements. For the execution of functional tests, knowledge about the inner structure of the SUT is not necessarily required. A systematic, planned, executed, and documented workflow has to be defined to successfully use functional tests [81, 421], including a test plan, test suits, test cases, and test verdicts [332].
> - ***Non-functional testing*** is similar to functional testing, but test cases are executed against NFRs to evaluate non-functional characteristics of the SUT.

The ***Level*** dimension has already been covered in the introduction of the chapter and refers to the scope of a test [260, 421]. The ***Technique*** dimension can be divided into the static and dynamic execution of tests.

Static testing techniques cover automated analysis, static analysis, and manual examinations, e.g., reviews, to analyze the source code of the SUT [258, 367].

Dynamic testing techniques can be categorized into white-box, grey-box, and black-box testing. Since dynamic testing is important in the context of this thesis, it is discussed in detail in Section 2.7.2 (p. 53 ff.).

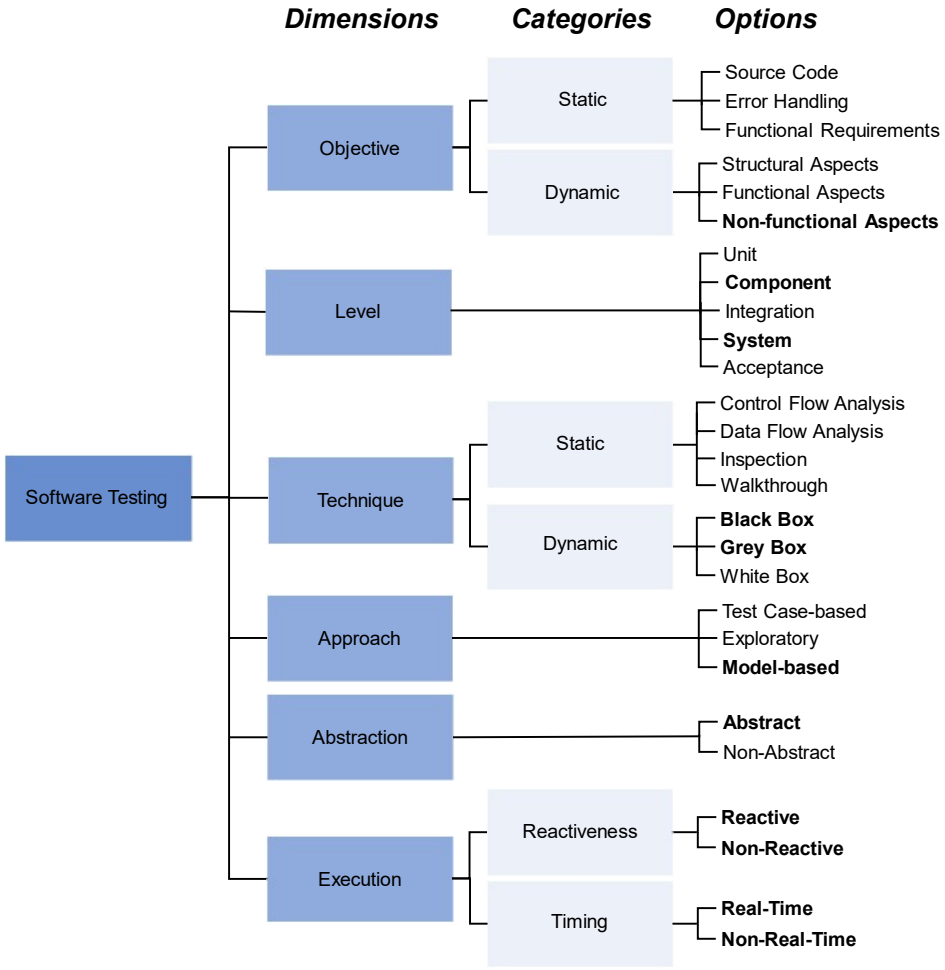


Figure 2.16: Dimensions of software testing, adapted from [260, 397, 421]. Dimensions may be divided in categories while options represent aspects of software testing. Options corresponding to the concepts presented in this thesis are highlighted in bold.

The ***Approach*** dimension describes the practice of how software tests are defined, represented, and executed as part of the test strategy. Figure 2.16 shows the most common approaches for software testing, such as test case-based testing, exploratory testing, and MBT. As a traditional method, test case-based testing refers to a practice where test cases are planned and defined in advance and derived from existing requirements. Instead of pre-defined test cases, in exploratory testing, the experience, knowledge, and intuition of software developers and testers are used to create test cases, also known as unscripted testing [6, 175]. MBT is a method in which software models are used as SUT and as a source for the test case derivation. This testing method is discussed in Section 2.7.3 (pp. 55 ff.).

The ***Abstraction*** of software testing refers to whether test cases can be executed directly or have to be transformed into executable test cases. UML sequence diagrams or a series of high-level procedure calls are examples of abstract test cases [397] that must be transformed into executable test cases [299]. Non-abstract or concrete test cases provide values for preconditions, input data, expected results, and a detailed description of actions [172].

The ***Execution*** of software tests shown in Figure 2.16 is limited to properties related to the concepts of this thesis, such as reactiveness and timing.

Reactiveness specifies if software tests depend on and respond to the behavior of the SUT to obtain results [172, 422], for instance, if data sent to the SUT depends on previous outputs. By this, the execution of reactive test cases is influenced by the behavior of the SUT, which allows test cases to be modified dynamically during execution. Reactive software tests require runtime monitoring of the SUT as described in Section 2.7.5 (p. 61 ff.). Since outputs of the SUT are used as inputs, reactive tests may be executed in a *closed loop*, while non-reactive tests are defined as *open-loop* tests. Open and closed-loop tests are elaborated in Section 2.7.4 (p. 57 ff.).

The ***Timing*** characteristic of test cases may include real-time aspects or requirements. Since time-related behavior strongly depends on the architecture, the test execution must be performed on the target platform or closely related prototype implementations under real-time conditions. If the requirements to be evaluated are not related to real-time or the execution time is not significant, test cases may be executed delayed.

2.7.2 Dynamic Testing

This section gives a brief overview of dynamic software testing methods. As mentioned before, software testing refers to the systematic analysis of software applications in controlled scenarios to discover errors and evaluate the behavior and quality level of source code w.r.t. requirements. Based on the availability of information about the SUT and the focus of tests to be created, a distinction can be made between black-box, grey-box, and white-box tests. Figure 2.17 illustrates these software testing methods related to their granularity level and required knowledge level.

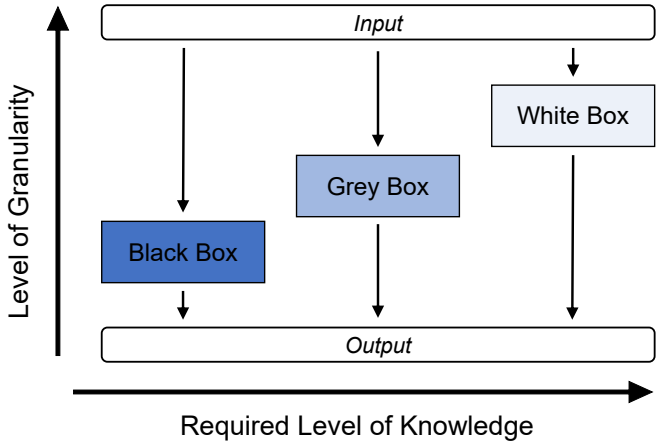


Figure 2.17: Black-box, grey-box, and white-box testing in relation to their level of granularity and required level of knowledge.

Black-box, grey-box, and white-box tests can be applied on different levels during software testing. In traditional approaches [59], black-box tests are used for acceptance and system tests, grey-box tests for integration and component tests, and white-box tests are mainly used for unit tests [406]. However, the assignment of these testing techniques to specific levels is not strict and may be adapted to the characteristics of the SUT, aspects to be tested (functional or structural), and the development process. As mentioned in [31, 57, 81], black-box techniques may also be applied on lower levels, e.g., as unit and component tests. In MBT, for instance, test cases are derived from models instead of source code, which defines MBT as a black-box testing approach [343, 397] applicable at different levels depending on the abstraction level of the model [306, 421]. Note that MBT can also be used, e.g., to automate white-box tests [421]. In the following, the three testing methods pictured in Figure 2.17 are introduced in more detail since they are part of MBD (cf. Section 2.7.3, p. 55 ff.) and XiL tests (cf. Section 2.7.4, p. 57 ff.).

White-box testing is a structure-oriented testing method used to evaluate the structure, source code, and control flow, e.g., functions and procedures, of a SUT, typically a software application or single software components [57, 235, 298]. Since test cases are derived from source code, white-box tests have the highest level of granularity. In order to specify white-box tests, developers must have in-depth knowledge of the programming language used and access to the internal structure and source code of the SUT. Due to the required knowledge, white-box testing is more extensive than other methods w.r.t. the selection of input data and the proof-of-correctness of the SUT [343]. White-box testing strategies include coverage-based tests such as statement coverage, branch coverage, and multiple condition coverage [57, 235, 258, 298, 406]. Regarding software testing levels (cf. Figure 2.15, p. 50), testing may be applied at the unit, integration, and system levels.

Grey-box testing defines a combination of white-box and black-box testing where an abstract model represents the system. While the internal algorithms and data structures are accessible to derive specific tests, inner low-level implementation details, such as the source code, are still hidden [35, 343]. Typically, grey-box tests evaluate the interaction between parts of the software application, e.g., on the integration level (cf. Figure 2.15, p. 50). According to Banerjee et al. (2016) [35], grey-box testing is closely related to MBT (cf. Section 2.7.3, p. 55 ff.). To derive and execute test cases, a model-generating process, a technique to generate test cases based on model explorations, and a test oracle are required. The test oracle compares the output of the SUT with the input defined by the test case to determine whether a test is valid or has failed. Possible model representations of the SUT may be timed state machines, e.g., Markov decision processes [148, 225], Markov chain usage models [311], and extensions such as timed usage models [357, 358]. Moreover, UML may be used for the model representation, e.g., based on component, object, class, use case, and state machine diagrams [35, 235]. In fact, Mall (2018) [235] describes grey-box testing specifically for UML diagram types, for instance, with state, state transition, and state transition path coverage for UML state machine diagrams and association, aggregation, and derived class tests for UML class diagrams. The test execution is similar to the black-box testing approach but with more specific test cases. Grey-box testing is typically applied on the unit, component, and integration levels [31].

Black-box testing corresponds to a functional testing method where the behavior of the SUT is evaluated. This method explores possible inputs and analyzes the corresponding outputs to validate that the SUT conforms to a specification. Operations provided by public interfaces can be used for all interactions with the SUT. At the same time, the internal structure

and implementation details of the SUT are hidden and unknown during the specification and execution of tests. Therefore, black-box testing requires the least level of knowledge and has the lowest granularity (cf. Figure 2.17). Test cases are derived based on requirements and interface specifications. The most common black-box testing methods are equivalence class partitioning and boundary value analysis [57, 184, 235, 258, 367]. Both methods aim to reduce the number of test cases by selecting only a few representatives of each class. Other black-box testing methods, such as error guessing, decision table, and pair-wise testing, are discussed in [81, 258, 367]. For UML-based models and model representations, the *UML Testing Profile 2 (UTP2)* provides a basis for systematic testing [31]. Usually, NFRs are evaluated implicitly when executing functional test cases on critical parts of the software application during black-box testing [367]. However, for the power consumption estimation approach presented in this thesis, access to inner details (grey box) is required during test case execution, e.g., to values of attributes and states of the software application. This topic is further discussed in Section 2.7.5 (p. 61 ff.). Black-box testing is typically performed in MBT [343], as discussed in the next section.

2.7.3 Model-based Testing (MBT)

Model-based Testing (MBT) is widely used in research and industry with slightly different interpretations. Utting et al. (2006) [398] introduce MBT as *"[...] a variant of testing that relies on explicit behavior models that encode the intended behavior of a system and possibly the behavior of its environment"* and define the term w.r.t. generative aspects as *"the automatable derivation of concrete test cases from abstract formal models, and their execution"*. Test cases may be described by pairs of input and output values. Schieferdecker and Hoffmann (2010) [343] describe MBT as a form of *"software testing where test cases are derived in whole or in part from a model that describes selected, often structural, functional, sometimes non-functional aspects of a SUT"* and define the concepts as *"the automation of the design of black-box tests"*. In [95], MBT is defined as a *"[...] testing technique aimed at the automatic generation of tests using models extracted from the software artifacts produced throughout the development process"*. For the automotive domain, Bringmann and Krämer (2008) [58] define MBT as *"all testing activities in the context of MBD projects"*. In [113], MBT is described as an *"umbrella of approaches that generate tests from models"*, and a more generic definition of MBT as *"testing based on or involving models"* is provided in [172].

MBT is mainly used for functional testing of the SUT and enables the execution of test cases at early development stages. Since the concept is based on models, software application tests are possible without the need for platform-specific source code to be available. Furthermore, MBT provides an efficient way to reduce the efforts to be made when defining and performing software tests leading to a reduction of costs for the overall software development process [343].

In the following, a taxonomy is introduced for a brief overview of the details and core elements of MBT. In [398, 397], the first comprehensive taxonomy for MBT was introduced and adapted in later work [399]. Additionally, it has been further extended for specific domains such as embedded systems in general [421, 422] and safety-critical hard real-time embedded control systems [29]. Figure 2.18 shows the basic structure of the taxonomy as a consolidation of the concepts presented in [398, 421]. The $A|B$ notation at the leaves indicates mutually exclusive alternatives. The taxonomy consists of the four main dimensions *Model*, *Test Generation*, *Test Execution*, and *Test Evaluation*, with eleven associated categories. It should be noted that the taxonomy of the category *MBT Basis* described in [397, 398] differs significantly

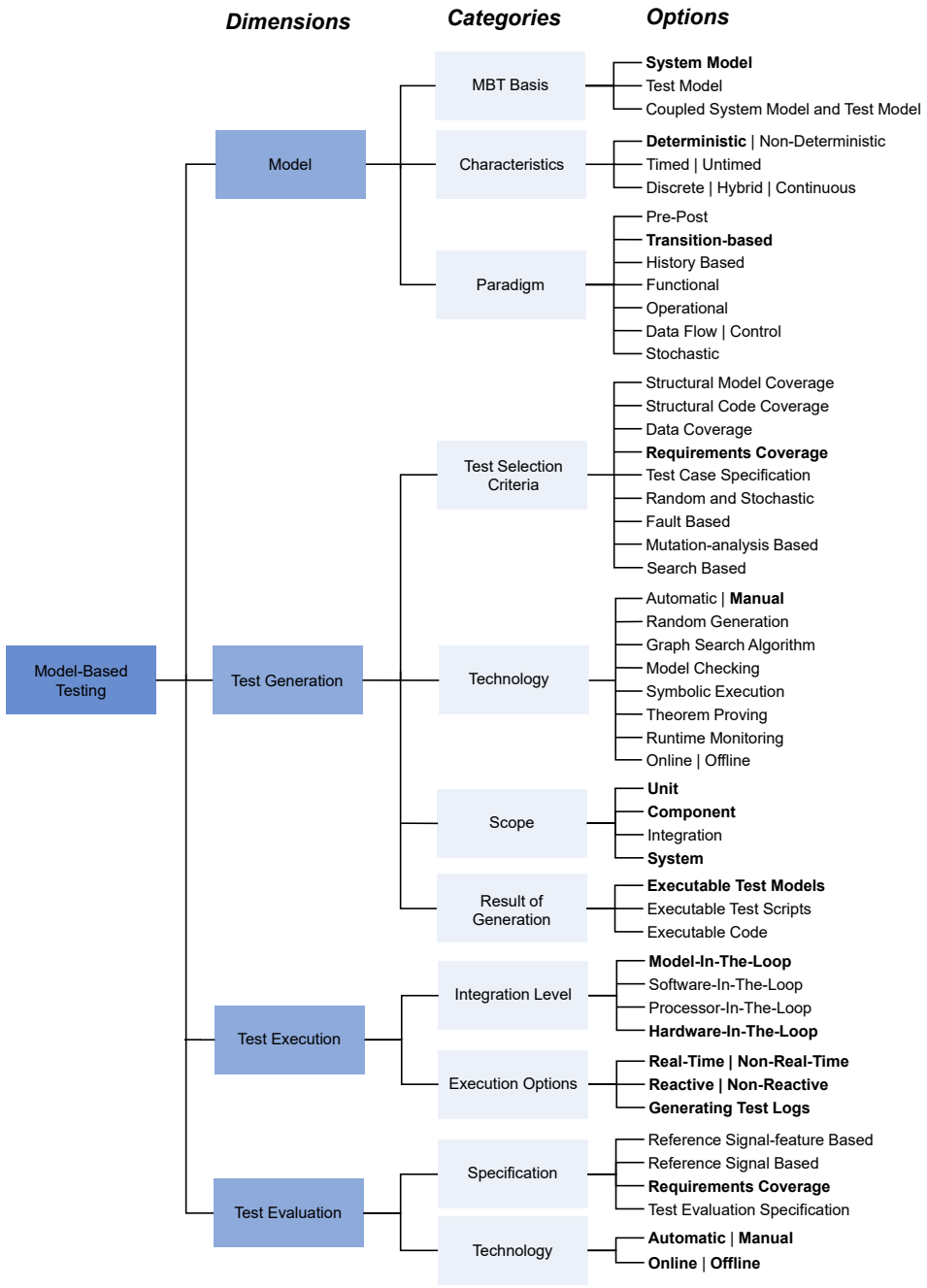


Figure 2.18: Taxonomy of MBT with dimensions, categories and options, adapted from [29, 398, 421]. Options related to the concepts of this thesis are highlighted in bold.

from [421]. Since the testing of environmental models is neither in the scope of this thesis nor relevant to the software engineering process, the definition introduced in [421] has been selected over [397, 398] as it is more suitable for this thesis.

Considering the definitions of MBT at the beginning of this section, the power consumption estimation approach of this thesis for MDD (cf. Chapter 5, p. 115 ff.) can be mapped to parts

of the MBT workflow based on the options of the taxonomy shown in Figure 2.18. As the MBT basis, a UML-based system model is investigated, which specifies the behavior of the software application extended with functional and non-functional aspects of hardware components. The system model has a deterministic discrete behavior and a transition-based notation, e.g., in UML state charts [275, 329]. Tests for requirements coverage are developed manually, resulting in executable test models. The test model can represent the complete software application or specific aspects, e.g., as UML sequence diagrams. The test execution is based on a system model and XiL approach. The power consumption estimation approach is used for NFR coverage performed online and offline in a manual or semi-automatic manner. In summary, parts of the concepts of this thesis can be integrated into the MBT domain based on the presented taxonomy. However, for the following reasons, the power consumption estimation approach for MDD does not aim to define an MBT approach. First, the automatic derivation and generation of test cases from models as a key factor of MBT is not part of the estimation approach and not covered by this thesis. Second, MBT has limited support for the generation of NFP-related test cases and the evaluation of NFRs [95, 96]. Even with UML extensions such as the UTP2 [276] to model test cases and environments, the capabilities to define non-functional tests are limited [31, 327]. Instead, a more general approach is provided to evaluate power- and energy-related aspects in early development stages of MDD based on a new UML profile (cf. Section 5.3, p. 124 ff.), methods (cf. Section 5.4, p. 135 ff.), and evaluation platforms (cf. Section 6.5, p. 161 ff.). The reader is referred to [397, 398, 421] for further information about the dimensions, categories, and options of the MDD taxonomy shown in Figure 2.18.

2.7.4 X-in-the-Loop (XiL) Testing

To evaluate software applications in MDD, *X-in-the-Loop (XiL)* testing can be used as a consistent testing process for the initial (partial) model, the generated source code, and the final software application executed on the embedded target system [59, 355]. The architecture of XiL tests can be divided into two main groups: open-loop and closed-loop tests. In open-loop tests, the SUT, e.g., the model of a software application, is executed in an isolated setup and evaluated without the impact of the environment. Inputs such as signals, parameters, or events are sent to the SUT, and outputs are monitored and analyzed. The general concept of an open-loop test [59, 186] is shown in Figure 2.19, where a test case is used to derive the test data as input for the SUT and the expected output of the test case. During test execution, the output of the SUT is monitored, captured, and compared with the expected output. This testing setup is denoted as an open loop since neither the output of SUT nor the behavior of the environment does affect the SUT and is returned as an input.

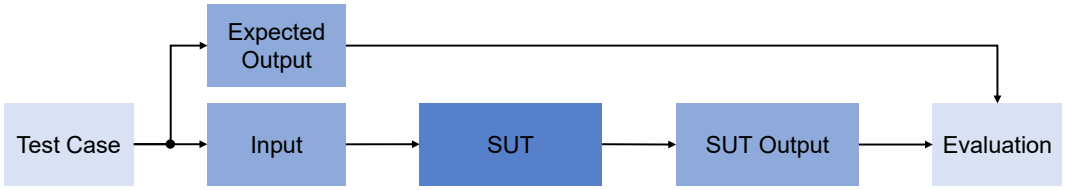


Figure 2.19: Diagram of an open-loop testing setup (block diagram notation).

For closed-loop tests, an environmental model, also known as *plant model*, has to be developed. It defines the behavior of the environment in which the SUT is tested. Instead of an isolated execution, both models are coupled, forming a closed loop [59, 203] that enables interaction between the SUT and the environmental model, as shown in Figure 2.20. The structure and scope of the environmental model may vary depending on the target domain, the defined SUT, and the abstraction level of the in-the-loop test used. If the SUT represents the software application of an embedded system, the environmental model can reflect a set of interfaces (e.g., sensors and actuators), define a set of physical properties (e.g., temperature and humidity), and simulate user-defined behavior. The test case is used to derive the expected output, input for the SUT, and the configuration of the environment model. The outputs of the SUT are used as inputs to the environmental model. The reaction of the environmental model is fed back as an additional input into the SUT, representing the idea of a closed loop between the SUT and the environment. An advantage of closed-loop tests is the ability to evaluate the interaction of the SUT with its environment before hardware platforms are available, which makes closed-loop testing suitable for early analysis.

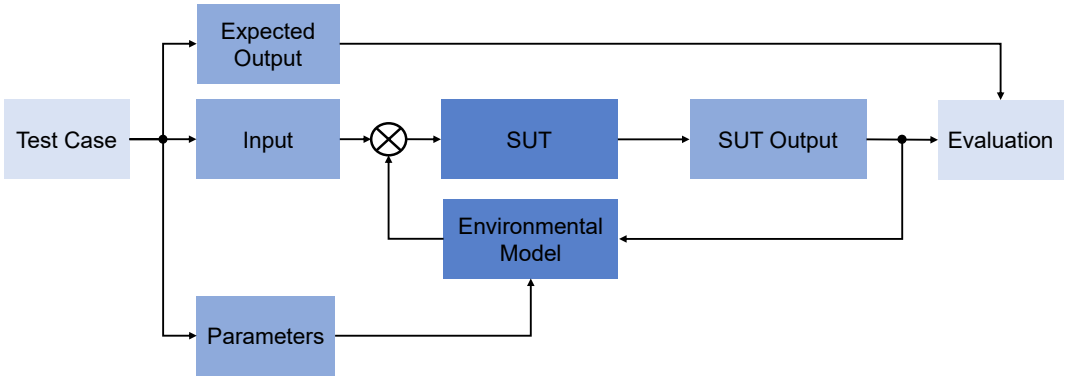


Figure 2.20: Diagram of a closed-loop testing setup (block diagram notation).

Open-loop testing is commonly used to evaluate the behavior of the software application model for static systems, which is equal to unit testing. In contrast, closed-loop testing is used for dynamic systems and focuses more on the interaction between the SUT and the environmental model.

Different in-the-loop test approaches exist, which can be used in various phases of the development cycle. This section uses the V-model development process[7] to illustrate in which development phases in-the-loop tests are typically performed and their time-related execution order. Figure 2.21 shows an abstract version of the V-model development process from a software development perspective for embedded systems with XiL tests assigned to the development phases. The V-model defines a standardized development process [404] as an extension of the waterfall model and describes the relationship between the development and testing phases. The y-axis defines the abstraction of the development process from defining requirements (high) to the design of the software application (low), while the x-axis defines the time of the development. The V-model is frequently used for developing software applications in domains related to embedded systems such as automotive [158, 249], space [255], or healthcare [242]. Note that different domain-specific definitions of the V-model

[7] In the literature, the term V-model is also referred to as V-cycle.

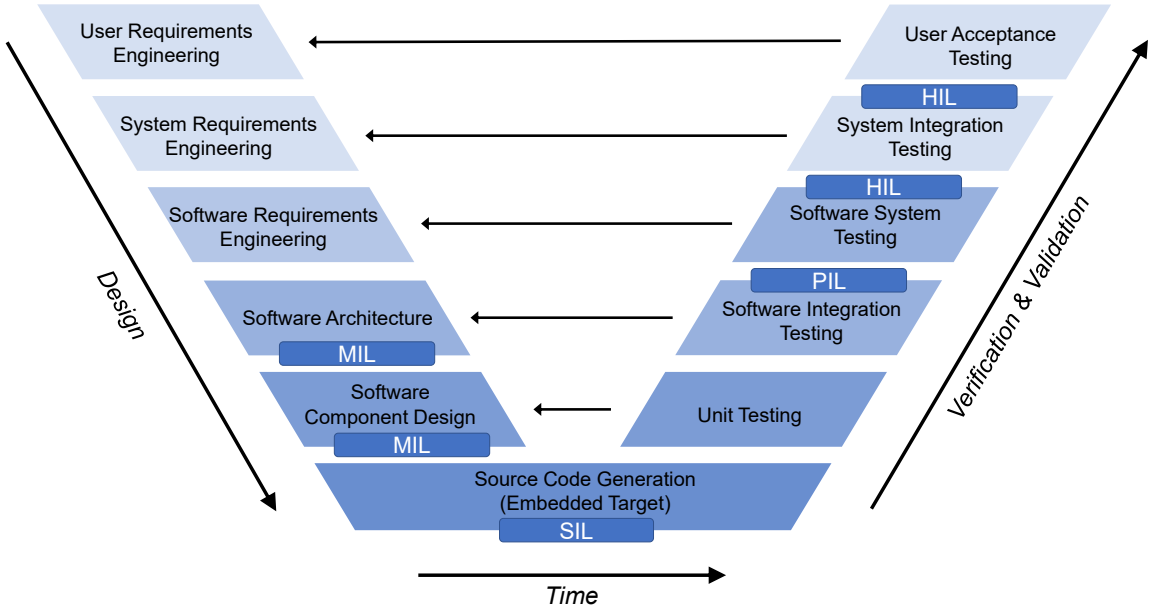


Figure 2.21: Illustration of the V-model with XiL tests to verify quality gates, adapted from [147, 158, 404].

approach instead of a uniform one exist [147, 404, 411]. Additionally, the V-model is the basis for other development processes. Examples are the multiple V-model [59] and the W-model [31, 343], which itself has different domain-specific interpretations, e.g., for software testing [368] and component-based software development [216]. Different levels of XiL tests, as shown in Figure 2.21, may be used to verify quality gates. A quality gate defines a decision point in the development process consisting of a set of quality criteria that must be met in order to enter the next development phase [59, 186, 355]. In the following, each in-the-loop test is explained in more detail.

Model-in-the-Loop (MiL) is the first level of in-the-loop testing, typically used in early phases of MDD to evaluate functional aspects of software application models. It aims to evaluate the architecture and functions to verify the developed software application regarding requirements defined in previous steps (cf. left side of Figure 2.21). To execute a MiL test, a runtime environment is required to provide the context for the SUT and to simulate other aspects required for the interaction with the software application model. For the simulation, laboratory and development computers are typically used. MiL can be seen as a hardware-independent testing approach for functional aspects and can be defined as open-loop or closed-loop environments. However, if MiL is also used to evaluate non-functional aspects, additional information about the embedded target system is required, e.g., for the evaluation or the environmental model. The testing process can be performed manually by developers or executed in a (fully) automated manner, e.g., using a model checker to inspect the software application model for compliance with the modeling guidelines and calculate the model coverage. Such tests can also be used to generate reference results for subsequent testing, e.g., *Hardware-in-the-Loop (HiL)*.

Software-in-the-Loop (SiL). In the next stage, the SUT is derived from the model representation and consists of hand-written or fully generated source code due to the use of code

generators. Broekman and Notenboom (2003) [59] refer to SiL as *host/target testing*, where the source code is compiled for the execution and evaluation in a simulated environment on the host computer. SiL tests can be executed in real-time using high-performance systems or fast and slow motion. Moreover, SiL may be applied in closed-loop and open-loop configurations with or without environmental models, e.g., to evaluate sequential and parallel aspects of the generated source code [102]. The primary use of SiL tests is to identify and fix inconsistencies in the functionality between the model representation and the executable object code. The automatic source code generation process requires a certain amount of preliminary work. The software application model may need to be adjusted using specific data types or libraries so that code generators can auto-generate platform-specific executable object code, which may impact the execution behavior. Design flaws, e.g., buffer overflows and arithmetic problems such as division-by-zero fault, are identified during SiL testing. Additionally, non-functional aspects such as memory usage or efficiency can be derived from SiL tests.

Processor-in-the-Loop (PiL). Described as software unit tests or software integration tests [59], PiL aims to test the real object code for the MCU that will later be used in the embedded target system. For this purpose, commercially available evaluation boards are typically used. Alternatively, target MCU emulators can be considered as an execution platform. As preliminary work, the cross-compiled source code is generated with predefined optimization options of the compiler and flashed on the target embedded system or integrated into the emulation platform. The embedded system, e.g., the evaluation board, executes the test management code as part of the PiL environment along with the compiled software application. The simulation tool performing the PiL executed on a host computer interacts with the management code, e.g., by using a serial communication interface, to send test values and obtain output results. Since the simulation tool and management code are involved in the PiL test, the real-time behavior of the software application cannot be evaluated. However, PiL aims to detect faults in the processor architecture and defects such as bugs in the code generator and compiler and undesirable side effects resulting from compiler options and optimization levels used. As the last level that provides inexpensive and manageable debugging during test execution [421], PiL fills the gap between the simulations-based testings (MiL and SiL) and the real-time level testing introduced next.

Hardware-in-the-Loop (HiL) tests are located in the last steps of the verification and validation process defined by the V-model shown in Figure 2.21. For HiL testing, the software application generated from the model is executed with a real-time operating system and other components such as network management stacks and drivers, e.g., for controlling peripheral devices and interfaces. Instead of an environmental model used in MiL, SiL, and PiL, a dedicated hardware setup (HiL simulator) specially designed to simulate the physical environment is required [259]. The HiL simulator has to be capable of performing actions in real-time, where a response must be guaranteed within specified time constraints. Since the environment can be simulated in real-time, the behavior of the SUT corresponds to the expected true behavior. Sensors, actuators, and additional components of the embedded system can be simulated or physically connected to the SUT as real components if no real-time models with the required accuracy are available [43, 77, 134, 173] or the creation of such models is too expensive or complicated [118, 232, 237, 390]. HiL testing aims to verify the correct functional execution of the software application on the target MCU while using the surrounding peripheral interfaces and connected hardware components [259]. Examples are scheduling misbehavior of the software application and performance issues due to the underlying hardware or latency issues based on the hardware-software interactions. Fault injections and edge cases can be

executed in a non-destructive manner to evaluate the safety and diagnostic functions of the SUT [259]. Furthermore, HiL simulators can simulate fluctuations and peaks of the supply voltage and peripheral misbehavior to evaluate the robustness of the SUT. By considering the entire embedded system, e.g., software and hardware, during the test case execution, HiL is the first level allowing the evaluation of NFRs, e.g., real-time performance [203]. Another use of application for HiL is the evaluation of interactions between embedded systems, which is why HiL can be executed in multiple stages of the V-model development cycle, as shown in Figure 2.21. It is also possible to achieve standardization and automatic execution of test series. However, HiL testing is considered to be time-consuming, expensive, and error-prone [319, 150]. The design of HiL tests requires a high effort while, at the same time, the physical SUT and the required environmental simulation are difficult to maintain. It also requires more time and costs to resolve bugs found at this stage of the development process compared to earlier phases, e.g., during MiL.

2.7.5 Performance Analysis and Runtime Monitoring

An important part of the software testing process is the execution and monitoring of the SUT, as well as the verification of compliance with requirements. As introduced in Section 2.2.2 (p. 24 ff.), software applications for embedded systems have specific requirements and unique constraints, which makes the validation process more challenging. The need to interact with the environment to obtain input data adds to the complexity of testing and validating software applications, as the environment may be dynamic, non-deterministic, and challenging to simulate or control [24, 35].

Performance analysis [84] is used to evaluate non-functional aspects of the software application (cf. Section 2.3, p. 27 ff.), which are typically related to the processing speed and response times of the SUT. However, power consumption as an important performance metric is often not associated with the software layer because a software application does not directly consume power [289]. As a driving factor of hardware activities, software applications significantly impact the dynamic power and energy demand of an embedded system since they define the actual behavior during runtime, which is also described as the cause-effect relationship [160, 231].

Non-functional testing is a central part of the testing process and differs significantly from functional testing. While many approaches exist to define functional tests, non-functional tests are difficult to describe without referring to functional behavior. Furthermore, NFRs are typically formulated abstractly and have to be converted into specific and executable test cases. According to Spillner and Linz (2021) [367], existing functional test scenarios are used to measure NFPs of the SUT for the verification of NFRs. However, the execution of performance tests, especially for estimating power consumption, is only considered useful if software applications have passed functional tests in advance.

The SUT is typically represented as a black box when evaluating non-functional aspects. To obtain the necessary data that test oracles may use for a test case evaluation, NFPs of the SUT have to be visible and measurable during the test run. For example, evaluating time characteristics, such as the response time as a non-functional aspect, does not require additional effort and can be measured and analyzed along with the execution of a functional test case. However, especially for estimating power consumption, an observation of the internal states is necessary, which causes the need for detailed and in-depth instrumentation and probing. As a consequence, the power consumption estimation approach requires the use of grey-box

tests. Nevertheless, observing internal states is not always a trivial task, especially for software applications used in embedded systems due to the lack of human-readable interfaces [133].

Runtime monitoring refers to an analysis paradigm where the execution of a SUT is observed and examined against the expected behavior [37, 117, 156]. It may be used as a practical application of formal verification and as an approach to complement conventional software testing and debugging [334, 412]. Besides the SUT, runtime monitoring requires a set of properties to be evaluated, which can be expressed, e.g., as a test using a formal specification language. According to Falcone et al. (2013) [116], the runtime verification process can be described as a three-step process:

1. Monitor synthesis
2. System instrumentation
3. Execution analysis

In the monitoring synthesis step, the monitor is generated, defining a runtime object used for the evaluation [116]. The monitor can receive traces, such as events or logs produced by parts of the SUT, and can evaluate the data considering the expected values [220]. The monitoring process can be performed during the execution of the SUT (online) or after the test run has been finished with log files (offline). Additionally, for online monitoring, the simultaneous execution of the test case and the monitoring can be synchronous, asynchronous, or a combination of both [72]. The synchronous approach defines a step-by-step analysis where a single event generated from the SUT is analyzed and processed by the monitoring instance before the SUT proceeds with the execution and generates the next event. In asynchronous monitoring, the SUT and the monitoring are detached and executed separately, which reduces the impact of the monitoring instance on the SUT and has lower overheads [71]. Additionally, the monitor is also able to send information to another source or back to the SUT if necessary. In terms of software testing, cf. Section 2.7 (p. 49 ff.), the monitor includes the role of the test oracle [220].

In the second step, namely the system instrumentation, the SUT is instrumented so that relevant events and signals from the SUT during runtime can be probed, extracted, and sent to the monitor. As instrumentation techniques, hardware-based and software-based instrumentation are used to obtain analog and digital signals of the hardware platform and generate the software application's output.

In the third step, the behavior of the SUT is analyzed by the monitor either online or offline, denoted as execution analysis. For the power consumption estimation approach, a hybrid monitoring approach of the SUT illustrated by Figure 2.22 is required to obtain data from the software application and hardware platform simultaneously during test case execution. Since runtime monitoring can also be applied after the SUT is placed in an environment, bugs in the software application that have not appeared in the testing phase may be detected [334].

The hardware instrumentation aims to measure the power consumption of the SUT during runtime. Data obtained by the software instrumentation may include extensive traces of hardware accesses, events, and signals [117]. Some variants of hardware-based instrumentation may require additional monitoring hardware (e.g., bus monitoring) or depend on specific hardware components (e.g., on-chip monitoring), which are out of scope for this thesis. However, further information can be found, e.g., in [412]. By combining hardware and software instrumentation, the monitor is able to correlate changes in current consumption with the behavior of the

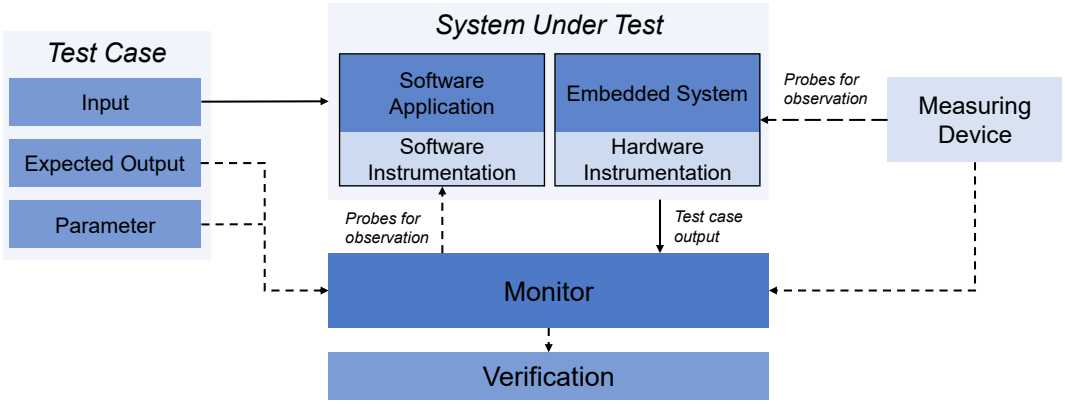


Figure 2.22: Runtime monitoring for the power consumption estimation approach, adapted from [412]. Dashed lines indicate information flow, e.g., logs, readings, expected output, and configuration parameters. Solid lines indicate data flow related to the test case execution.

software application. Sánchez et al. (2019) [334] published a survey on challenges for runtime monitoring. For the power consumption estimation approach presented in this thesis, the challenges of observability and non-intrusiveness are particularly significant. An ideal monitoring implementation should not interfere with the behavior of the SUT used for test case execution. However, power consumption can be measured without interfering the SUT during runtime, which makes the hardware instrumentation in this thesis inherently non-intrusive.

The software instrumentation is executed along or included in the software application of the SUT, which can affect functional and non-functional aspects and, thus, the behavior of the SUT during runtime. Consequently, the result of the test case execution may be different compared to the execution without instrumentation. This effect is also known as the observer or probe effect [129, 367]. Examples of such negative effects may include a higher response time or a more intensive hardware utilization, which in turn affect the power consumption estimation. Therefore, it is common practice to use software instrumentation with care while aiming for low latency to keep the effect as minimal as possible [334].

2.7.6 Related Work on the Integration of Virtual and Physical Hardware

This section discusses related work considering the integration of virtual and physical hardware for simulation and testing. For the power consumption estimation of software applications in MDD, the hardware behavior must part of the simulation to reflect hardware-software interactions due to the cause-effect relationship between software and hardware. The proposed power analysis methods (cf. Section 5.4, p 135 ff.) introduce concepts to couple and integrate virtual and physical embedded systems into the simulation environment for a system-wide power estimation in early development phases. Previous research has proposed approaches related to the basic conceptual ideas of the proposed power analysis methods but with a different focus and only slightly related to topics such as MDD, software testing, the modeling of energy and power-related aspects, and the influence of the software application on energy consumption.

An approach on the program level to automatically determine the energy demand of an MCU is presented in [161]. The authors introduce *indirect* and *direct* energy measurement

techniques to give developers energy consumption measurements and optimization hints at the function level. In contrast to the approaches presented in Sections 5.4.1 to 5.4.2, the source code must be compiled and flashed. The concept of *direct* measurement is somewhat similar to the power analysis method based on the integration of physical embedded systems presented in Section 5.4.2 (p. 136 ff.), which describes the use of a measuring device to obtain measurements. The *indirect* measurement in [161] uses performance counters to determine the energy consumption of the software application directly executed on the target platform. The work aims to optimize software applications on the source code level and to detect the best suitable compiler and compiler flags. The authors also state that their approach is used in early stages of the software development process. However, in MDD, source code-level approaches are no longer considered as early. Suppose further peripheral devices in addition to the MCU are included in the evaluation process. In that case, a full-blown platform-specific software application has to be available. Additionally, the approach in [161] only considers the energy consumption of the MCU and does not take other components, such as wireless interfaces or sensors, into account.

In MBD, co-simulation is an area of research where frameworks are used to couple simulation environments and models executed on different hosts, which may be placed in various geographical locations. Protocols such as the *Distributed Co-Simulation Protocol (DCP)* [254] have been specified for the interoperability and communication between distributed tools and simulated models. The DCP implements a master-slave principle and specifies a data model, a *Finite State Machine (FSM)* [410], a set of protocol data units, and a communication protocol that supports IPv4, *Controller Area Network (CAN)*, USB, and Bluetooth for data transport. For each system involved in the communication process, a particular DCP slave has to be executed on the target host in addition to the simulation. DCP has been used in related work, e.g., in [39, 209, 346], to achieve XiL tests. A co-simulation scenario with a remote embedded platform has been presented in [346]. However, embedded systems do not always provide interfaces such as Ethernet. In addition, wireless communication interfaces are not suitable as an interface between simulations for a power analysis due to their high power consumption, which would negatively affect the overall measurement of the embedded system. Furthermore, the embedded system used in [346] is not considered resource-constrained, which makes the approach not suitable for the requirements of this thesis.

A model-based rapid prototyping process is presented in [305]. The authors describe the integration of software and hardware models based on a model-based integration environment and a newly defined graphical architecture description language [304]. The language is a subset of Simulink and Stateflow and provides time-triggered semantics, which restricts the functionality of software applications to periodic executions. The hardware platform is abstracted and virtualized while supporting communication interfaces to collect sensor data and provide signals. However, the approach uses Simulink and Stateflow while this thesis is focused on UML-based models. Furthermore, the authors do not consider NFPs in the software and hardware model description, which makes the rapid prototyping process unsuitable for an early power consumption estimation.

A MiL framework to validate algorithms for autonomous driving and advanced driver assistant systems is presented in [61]. As part of the proposed approach, a hardware-agnostic middleware is used as a communication framework to interconnect data sources, e.g., sensors, the algorithms under test, and the vehicle model. Inputs are collected from either real or simulated virtual sensors. Although the work in [61] defines a MiL approach, it differs significantly from the approaches presented in Sections 5.4.1 to 5.4.2. The approach aims to perform func-

tional tests for algorithms written in C++ instead of non-functional tests of software models or artifacts in the context of MDD. Additionally, the framework is not focused on energy and power-related aspects. The middleware may provide data from actual sensors. However, the algorithms do neither directly nor indirectly interact with the data sources, e.g., sensors. The algorithms are platform-independent, while all data sources and the vehicle model are defined within the same physics-based simulation platform, which makes the middleware proprietary. Furthermore, no real sensors were used for the evaluation.

In [365] and follow-up work [366], the authors present a robot simulation and monitoring system. Their approach is able to communicate with simulated and real devices similar to the approaches presented in Sections 5.4.1 to 5.4.2. However, while proposed power analysis methods (cf. Section 5.4, p. 135 ff.) aim to provide an environment for early testing and evaluation of software applications, the work proposed in [365, 366] provides a *Graphical User Interface (GUI)*-based solution for remote control of robots by users. Additionally, the authors do not consider aspects related to energy and power.

2.8 Related Work on Power Consumption Modeling and Estimation

This section discusses work in the area of power modeling and estimation that is related to the overall concept of this thesis. In [143, 375], the authors state that optimizations on higher abstractions levels for hardware and software lead to significantly higher power savings and faster estimations. Figure 2.23 illustrates different abstraction levels of hardware and software design. In MDD, models inherently represent the software application in an abstracted sense on the architecture and behavior levels and are transformed by, e.g., model-to-text transformations (cf. Section 2.5.2, p. 40 ff.), into program code. As shown by the left y-axis of the diagram in Figure 2.23, power savings due to higher-level changes are expected to be orders of magnitude higher than lower-level changes. Moreover, the time required to perform estimations is expected to be orders of magnitude lower for each level of abstraction (right y-axis in Figure 2.23).

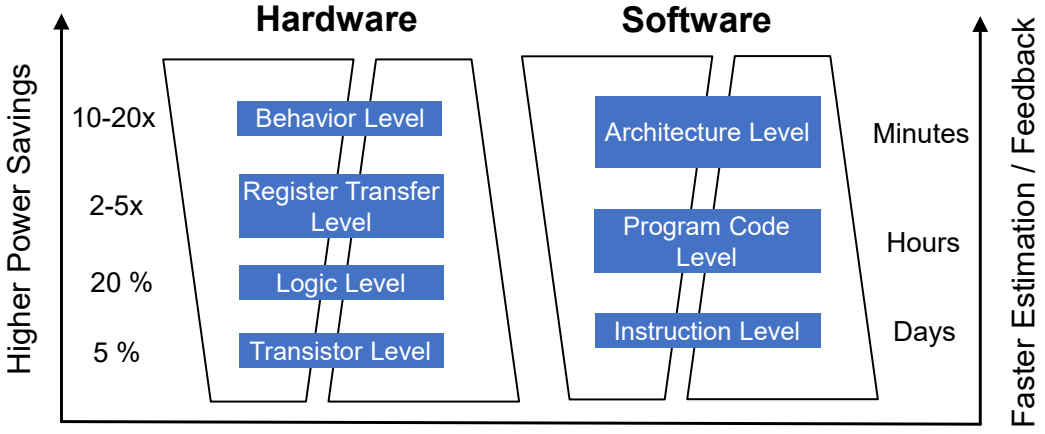


Figure 2.23: Power reduction and estimation efficiency for different hardware and software levels, adapted from [375].

As illustrated by Figure 2.23, the power consumption estimation can be addressed at various levels of hardware and software design. Related work is summarized and compared in Figure 2.24 based on their ability to consider the entire system (*system view*), e.g., hardware and software aspects, as well as the *level of abstraction* and, thus, their *suitability* to be used in MDD, specifically in early development phases. In Figure 2.24, groups of thematically similar approaches are indicated by a specific color. The composition of the groups is discussed below. This section does not distinguish between related work on power modeling and estimation since most research provides both within the same work.

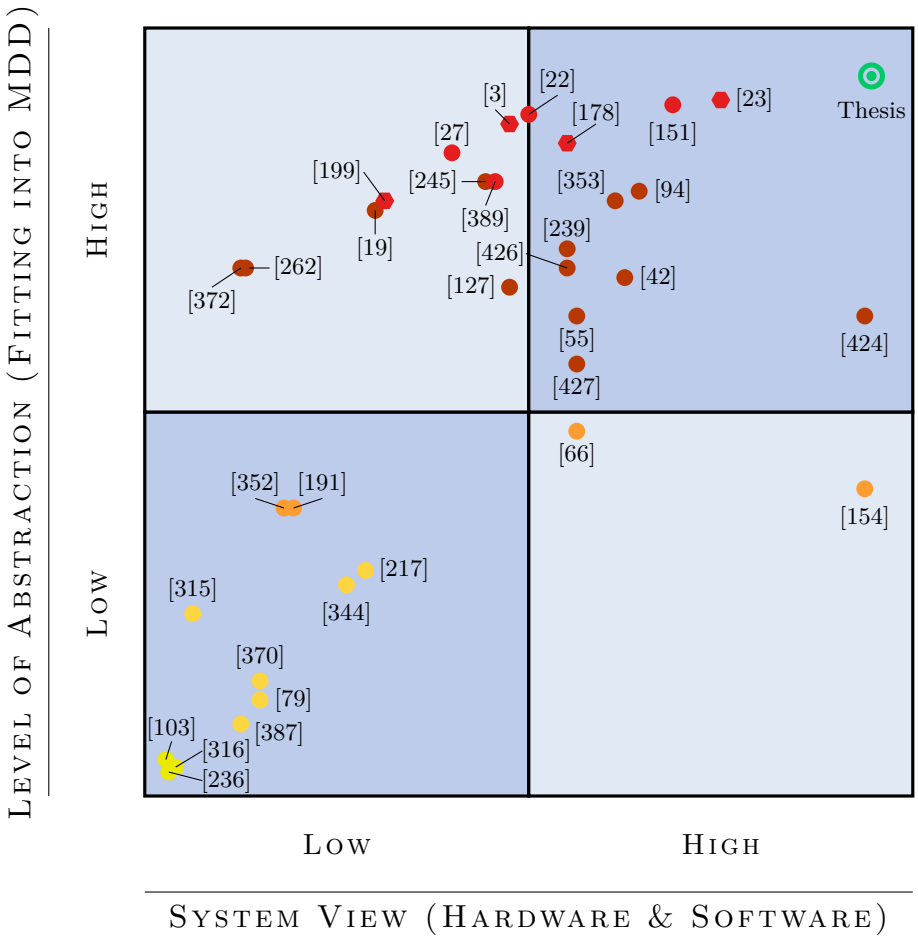


Figure 2.24: A 4-quadrant diagram comparing related work in power consumption modeling and estimation according to the system view and level of abstraction criteria. Thematically similar work is grouped together. A distinction is made between low-level (yellow), code-based (dark yellow/orange), various model-based (brown), and UML-based approaches (red). Hexagons describe work with a stronger focus on software. The approach of this thesis is shown in green.

2.8.1 Low-level and Source Code-based Approaches

Low-level approaches at the transistor level, gate level, and *Register Transfer Level (RTL)* [103, 236, 316] offer highly accurate simulations but require a deep understanding of the internal structure of the hardware component to be modeled. Due to the complexity, these techniques are often used to simulate single components or *Intellectual Properties (IPs)* of a component. Provided approaches, e.g., [103, 236, 316], can be considered very accurate with low overhead. However, existing models mainly cover parts of the CPU and, thus, address only a specific part of embedded systems. Low-level approaches are also unsuitable for complex structures, including multiple hardware components. In Figure 2.24, low-level approaches are colored yellow and located in the bottom left corner of the diagram.

Instruction-level Power Analysis (ILPA) and *Functional-level Power Analysis (FLPA)* are higher-level approaches compared to transistor level, gate level, and RTL approaches. In ILPA, the power consumption is estimated based on instruction and instruction pairs [79, 387, 370]. ILPA and FLPA are located on the assembly level, while the FLPA concepts also have been applied to algorithms written in the C programming language [191]. The energy cost for each instruction has to be measured in detail using experimental environments of the CPU, which can quickly become unmanageable for complex architectures with large instruction sets. FLPA decreases the time needed to build energy models and has been first mentioned in [315], where basic instructions, built-in library functions as a sequence of basic instructions, and user-defined functions as a combination of both are used to estimate power consumption. More recent variants of FLPA, however, are focused on the functional level of the CPU itself. For this, the CPU is modeled as a block [191] or divided into multiple functional blocks [217, 344, 352], while the consumption depends on algorithmic and configuration parameters. Energy models are typically based on equations or represented as tables providing the consumption for each instruction. Approaches such as ILPA and FLPA primarily focus on estimating the power consumption of CPUs. They are not intended to be used for a system-wide analysis where software applications interact with other hardware components. Hao et al. (2013) [154] proposed a static analysis of Java byte code extracted from smartphone applications to estimate the overall energy behavior of mobile applications. The energy model used is based on per-instruction energy cost functions for each hardware component of the device, such as the MCU, RAM, and wireless communication module. For the estimation process, the whole application, execution paths, or single methods may be used. The effort required to create the energy models based on instruction-accurate measurements for various hardware components should not be underestimated. However, further details on the energy model are not provided in [154]. Finally, the need for hardware-specific assembly code or source code in low-level languages such as C makes those approaches unsuitable for power estimation in early design phases for software applications in MDD (cf. Section 2.5, p 37 ff.), mainly since source code is automatically generated at later stages of the MDD approach. Still, approaches such as [154, 161] may be used in later stages to optimize generated source code on the task level considering MCU-specific properties.

In [66], a concept for an energy-aware device driver is presented, which contains a power-annotated state machine to trace and map the current state of the hardware component to a driver state for an overall power consumption estimation. The concept presented in [66] uses the state machine model of a hardware component to generate function signatures for an energy-aware device drivers. Software developers have to manually refine each of the generated functions afterward. Although the basic idea to trace software-hardware interactions as the

behavior of the software application is similar to the approach presented in this thesis, their concept does not relate to MDD, where standardized modeling languages for the description of state machines and NFPs are used as an integrative part of software application models. Instead, drivers with hardware-specific aspects must be directly implemented on the source code level to enable power consumption estimations. Furthermore, the proposed power model is based on scalar values and does not support parameters that can be influenced by the software application during runtime. Although device-specific source code generation is not in the scope of this thesis, the approach in this thesis may be used as a basis for generating energy-aware device drivers.

Since the aforementioned approaches are based on second-generation and third-generation languages such as assembly and C, the level of abstraction is low, and the approaches do not fit into MDD. Regarding the *level of abstraction*, ILPA, FLPA, and source code-based approaches are located in the lower part of the diagram presented in Figure 2.24. However, some approaches consider multiple hardware components so that the *system view* may vary. Approaches based on assembly language or lower are colored in a darker yellow tone, whereas approaches using more high-level languages are highlighted in orange.

2.8.2 Model-based Approaches

Power and energy models for embedded systems are typically considered on the hardware component level, with individual models for each component. With a focus on power consumption modeling and estimation, the following related work provides concepts for hardware component and system modeling along with NFP and NFR modeling approaches. In Figure 2.24, the introduced work is illustrated with a brown color and combines model-based approaches of different domains, e.g., mathematical, SystemC, or *Architecture Analysis & Design Language (AADL)* models.

Concepts for power consumption modeling and estimation have been proposed in a set of research approaches based on workflow models [427] with stages of the system (e.g., initializing, sensing) that specify the power level of each modeled hardware component, Petri nets [19] to represent the power-related behavior of the system, and high-level mathematical models [55, 239, 426] to address power estimation on higher abstraction levels. In addition, manufacturers of hardware components also offer tools that developers may use to estimate the power consumption of hardware components. For instance, as part of the STM32CubeMX graphical configuration and low-level code generation tool published by *STMicroelectronics* [372], the *Power Consumption Calculator* provides a power estimation for MCUs. With this tool, developers can define profiles based on power state sequences with hardware-specific configurations. Similar approaches are available for other hardware components. The company *Nordic Semiconductor* offers an *Online Power Profiler* [262] for their nRF52 Bluetooth low-energy and nRF91 low-power wide-area network radio communication product lines. The mentioned tools use mathematical models based on measured values to estimate power consumption. Measurements are typically derived under ideal conditions in a controlled test environment, which may not be reproducible in a real scenario. Other work, such as [42], focuses on the system level, where, for example, multiple components are modeled as functional blocks and state machine representations. However, none of the mentioned related work does consider the integration into MDD and the impact of software applications.

In [127], an algorithm for the automatic generation of parameter-aware energy models of single hardware components for embedded systems is presented. An energy model is represented

as a state machine in which each state is related to a distinct state of the hardware component. Transitions correspond to functions of a simplified driver or interrupts. The approach is not related to MDD, and generated energy models are not described with a standardized modeling language such as UML. However, the proposed approach for generating energy models may be used as a source for hardware component models (cf. Section 5.2.2, p. 119 ff.) by applying appropriate model-transformation techniques described in Section 2.5.2 (p. 40 ff.).

Zhang et al. (2010) [424] describe a concept to estimate the power consumption for smartphones executing compiled applications. The power model is represented as a linear equation with zero-one indicators derived from traces of hardware components. In contrast, this thesis aims to provide a power consumption modeling and estimation approach as part of the MDD process, where software applications can already be evaluated in early development phases without requiring compiled source code. Additionally, mathematical models proposed in [424] can be integrated into hardware component models using UML state machines and VSL to model NFPs as extensions of state machine descriptions.

A MDA framework to estimate the power consumption of *Near Field Communication (NFC)* devices and to verify power-related NFRs is presented in [245]. The proposed models are based on SystemC [170], where each module (e.g., hardware component) is extended with a power state machine. Furthermore, use cases defined as sequence diagrams represent the software application. Although the basic idea of integrating physical hardware into the simulation process is similar to the approach of this thesis, the presented power consumption estimation is limited to use cases for the communication between the NFC-Reader and NFC-Bridge, supplied by the NFC-Reader during communication. In contrast, the approach of this thesis is designed to consider the complete system. A more fundamental difference compared to [245] is that the approach of this thesis aims to evaluate software application models instead of hardware components (e.g., NFC-Readers) that supply power to and interact with existing systems.

Another approach for a system-level power and energy consumption estimation based on the AADL is presented in [94, 353]. The authors focus on real-time operating systems, *Field-programmable Gate Arrays (FPGAs)*, and data transfers for the Ethernet communication of client-server architectures, for which an estimation approach based on power models presented in [93] is provided. In AADL, software components consist of data, threads, and process components and are bound to hardware platform components (e.g., CPU or buses). As part of the estimation process, software and hardware platform components are extended with *power capacity* and *power budget* properties, calculated in a two-step process. For instance, a bus system may have a *power capacity* property while every hardware component accessing the bus declares *power budget* draws from the bus. In the first phase of the process, the *power budget* for every software component (thread) is computed as an power estimation. In the second phase, the *power budgets* are combined to calculate the *power budgets* for every hardware component as the basis for the energy analysis. The basic power model for each hardware component is defined as a set of consumption laws for different parameter combinations. Instead of AADL, this thesis focuses on UML as the most used modeling language in the embedded software industry [7] and relies on UML class and UML state machine definitions. By using the same modeling language and elements as the software application to describe hardware components, the analysis is eased since the system is represented by a single model in the same representation. The PAP presented in this thesis (cf. Section 5.3, p. 124 ff.) uses MARTE, which is also compatible with AADL component models [278, 119]. Additionally, embedded systems used in IoT and IIoT are typically not based on FPGAs. The consumption laws

specified as linear equations in [94, 353] are not integrated into AADL models. In contrast, the approach in this thesis provides model annotations for all power and time-related properties and methods to define the dynamic behavior of hardware component models enabling a model-to-model transformation without information loss.

2.8.3 UML-related Approaches

The following approaches are based on UML, SysML, and partly on MARTE, which makes them more related to the concepts presented in this thesis. Related work discussed below is colored red in the diagram shown in Figure 2.24. To outline related work with a stronger focus on software applications and, thus, with the closest relation to the approach presented in this thesis, a hexagon is used as an illustration.

A model-based power consumption analysis technique based on UML models is presented in [199]. The introduced systematic process transforms the software application modeled with UML-based sequence diagrams into control flow graphs. Each node of the control flow graph is associated with a set of virtual functions (primitive instructions and system functions) for which an energy measurement must be performed in advance. However, the energy analysis is still code-based, even if a model-based approach with UML as the source is used. Basic instructions and functions abstracted in the model are defined by an action language (cf. Section 2.5.1, p. 38 ff.). The approach presented in [199] may provide an earlier power consumption analysis compared to other code- or function-based approaches mentioned above. However, it focuses exclusively on the MCU and does not consider the overall system.

Trabelsi et al. (2011) [389] propose a model-driven hybrid power estimation for embedded systems architecture and application models in UML, extended with MARTE. Their work focuses on the CPU, cache, shared memory, and communication buses as part of an SoC and provides power estimation modules for black-box and white-box IPs. The simulated architecture and the power estimators are generated automatically using an MDE approach, while the estimation is still based on cycle- and bit-accurate simulations in SystemC [170]. However, the approach presented in [389] is not intended to consider a complete embedded system. Additionally, if the system consists of multiple complex components, e.g., dedicated sensors and wireless communication modules, an in-depth analysis of each component and sub-component must be performed to derive the behavior of internal memory and communication buses. Furthermore, their proposed energy estimation approach is only suitable for offline estimations. While this thesis focuses on software models and hardware-software interactions for the estimation process supporting online and offline estimations, their work is more concerned with the design space exploration of SoCs.

In [27], for instance, a low-level device modeling and high-level power estimation solution are described. The approach uses the IP-XACT standard [171] to describe hardware models, while UML state machines and UML profiles define the appropriate power-related behavior. In MDD, models of low-level hardware components (e.g., clock generators) are not suitable for evaluating software application models in early design phases due to the high complexity and amount of sub-models needed to specify a hardware model of a sensor or MCU, especially if the analysis has to take the interaction and impact of the software application on single components (e.g., sensors) or the entire system into account. Moreover, the approach does not consider the dynamic behavior of peripheral devices. For example, environmental or configuration changes can cause the peripheral device to consume different amounts of power in an operating or power state.

In [178], an approach for energy-aware scheduling and timing analysis of software applications is presented. The concept is based on a timing-energy analysis model obtained by reverse engineering processes of existing source code resulting in UML class representations [183]. However, the timing-energy analysis model does not include any implementation details. As a result, the evaluation is based on task level and predefined execution times. While the approach of this thesis is able to take dynamic behavior and multiple hardware components into account, the work in [178] focuses on analyzing the power consumption of the MCU.

An approach for power estimation and optimization of CPUs using UML profiles is proposed in [3]. The application is modeled using tasks and UML activity diagrams. The CPU model consists of a state machine with associated operating modes, defined as a tuple of voltage and frequency values and thresholds for over- and under-utilization. For each additional operating mode mapped to a logical state of the CPU (e.g., execution, sleep, and idle), an additional state has to be defined, which makes the approach prone to the state explosion problem [342]. State transitions are executed at the start or end of a task based on the current utilization of the CPU. While the approach presented in this thesis focuses on the application level and interactions between software and hardware components, their work is concerned with finding the best CPU utilization while not taking any other system components into account. Additionally, the approach does not consider the dynamics originating from software applications, and no measurements were performed to validate the results.

Another profile based on UML and MARTE for power consumption and real-time analysis of embedded systems is introduced in [151]. Stereotypes of the proposed profile contain tagged values to model the switching capacitance, leakage power consumption, and voltage-frequency pairs used to distinguish between working modes of CPUs. Battery models are extended with voltage and capacity descriptions, while other component models, like displays, are limited to their static power consumption. To consider the software application, tasks without implementation details are defined. Instead, the characteristic of a task is defined by proper stereotypes, which provide the execution interval, worst-case execution time, and worst-case execution cycles. Tagged values are extracted and used in simulations to find the most power-efficient task-by-task working mode for a CPU while considering DVS and real-time requirements for every task and task chain. However, while the approach in [151] focuses on simulating task execution on CPU models based on their modeling timing characteristics, the approach presented in this thesis is designed to take software application models for the power estimation process into account. By providing stereotypes for class and state machine diagrams, both static and dynamic aspects of hardware components can be considered.

As an extension of MARTE, the authors in [22] and follow-up work [23] introduce a profile to model system-wide dynamic power management aspects of embedded systems. A state machine is defined for each hardware component to model their behavior. States represent operating modes and are extended with power-related aspects, e.g., frequencies, static power consumption and maximum energy per cycle, provided by the defined UML profile. The system-level view is defined by power configurations. A power configuration is associated with a system-level power mode and specifies a set of power states of hardware components to be active within the specific system state. The concept introduced in [23] provides two use case scenarios for the proposed UML profile. The first scenario defines a power state machine for a many-core CPU, including states for each core configuration with associated power configuration. Based on over- and under-utilization measurements, the number of active cores can be adjusted during simulation. In the second scenario, the application is modeled as use cases directly bound to power configurations. The concept of dynamics is limited to

the idea that hardware components may have different states depending on the current use case. Besides the MCU, only a video codec accelerator is considered. However, configuration details of hardware components are only addressed slightly.

2.9 Summary

This section concludes the introduced background and related work to discuss identified gaps w.r.t. each of the four RQs introduced in Section 1.2.2 (p. 8 ff.). Furthermore, methodological, technological, and design choices based on findings are elaborated.

RQ1 – Formal Definition of Energy-related Behavior and Defects

RQ1 aims to answer how NFRs may be described to specify energy-related behavior. For this, it is necessary to revise the definition of energy bugs and define a more general classification. To prevent a violation of power- and energy-related NFPs from being interpreted as an energy bug instead of an unconsidered behavior, an additional definition of the environment is required, which is used along with the formal specification of NFRs during testing for the detection of energy bugs in a more precise manner.

The related work and findings introduced in Section 2.3.3 mainly focus on smartphones [34, 83, 101, 221, 293, 424] as only one possible embedded system. Some related work [34, 264, 265] analyzed the primary sources of energy consumption, denoted as energy hotspots. However, energy hotspots do not represent bugs per se but may indicate abnormal functional behavior. The introduced taxonomy and classification of energy bugs in [34] are insufficient as a basis for the specification of extensive power- and energy-related NFRs by embedded system designers and engineers. First, software applications for constrained embedded systems use special-purpose and resource-efficient operating systems with less functionality than mobile devices such as smartphones. Second, the precise control of individual sub-components of the system is not reflected in the classification mentioned in [34]. Some types of energy bugs discussed in [293], such as the *wireless signal strength*, describe the behavior of hardware components, occurring rather unexpectedly than unintentionally, which does not fit into the definition of a traditional bug. Furthermore, energy bugs discussed in [34, 226, 293] mainly refer to effects without providing a quantitative definition that can be expressed as an NFR and used directly as a criterion in testing. Due to the significant influence of the environment on the behavior of embedded systems used in IoT, properties that can influence energy behavior must also be considered. Such properties include parameters and configurations of sensors and actuators as a direct interface to the environment and parameters of the environment itself to avoid false-positive detections of energy bugs that only indicate unconsidered behavior.

RQ2 – Best Practices and Design of Energy-aware Software Applications

RQ2 aims to answer how energy-aware software design patterns are described uniformly. To address RQ2, the analysis of related work emphasizes the need for a novel framework to describe energy-aware software design patterns in a more general and quantifiable manner. By this, a direct comparison between different design patterns may be achieved, which helps developers to select the most suitable solution.

Related work presented in Section 2.4.2 (p. 34 ff.) discusses power- and energy-related aspects of software design patterns from two perspectives. The first perspective discusses

related work on *improving* software design patterns by optimizing or developing alternative solutions for language-specific design pattern implementations [64, 122, 224, 234, 263, 331]. However, the *improvement* of design patterns is not in the scope of this thesis. The second perspective contains related work on domain-specific design patterns *enhancing* software applications addressing NFPs such as safety or power and energy consumption. This is achieved by integrating the impact on NFPs as the subject of NFRs into the design pattern template and, thus, describing the consequences and impact on NFRs. Alternative design pattern templates considering consequences and effects on requirements, NFRs, and constraints have been presented in [21, 204, 321, 322, 325]. While some design patterns target the architecture of the ecosystem [321, 325] in which an embedded system may operate, other approaches strictly divide design patterns into software- and hardware-based patterns [21]. However, mentioned related work does not consider power- and energy-related properties. In [85, 86, 228, 308, 309, 323], the authors propose design patterns that may be used to develop more energy-efficient software applications. However, only part of the related research, e.g., [85, 86, 228, 323], describes design patterns addressing power and energy aspects of software applications executed on embedded systems. To the best of our knowledge, the design pattern templates discussed do not integrate the impact on power consumption and time behavior as two closely related NFPs, in their structure and fields. They also do not provide a quantification of the energy-related cause-effect relationship between software and hardware. The description of the design patterns is only expressed in natural language without providing information about the impact on power- and energy-related NFPs and NFRs. Therefore, developers cannot consider the impact, side effects, and tradeoffs before applying a specific design pattern. Moreover, as stated in [38], there is still no catalog of design patterns targeting energy efficiency in software engineering. To demonstrate the applicability of the introduced framework, a first catalog of energy-aware design patterns based on the introduced design pattern template is presented in this thesis.

RQ3 – Joint Modeling of Functional Software Application Models and Energy Behavior

RQ3 deals with software application models and how they can be extended with energy-related hardware characteristics to make the software-related impact visible and traceable. An analysis of related work indicated a lack of modeling approaches for proper power consumption estimation of software applications. To answer RQ3, concepts are required to model power-related properties of hardware components and define the dynamic power-related behavior of an embedded system. By this, the cause-effect relationship between software and hardware may be considered and analyzed to estimate power consumption.

UML is selected as a basic modeling language since it is well-suited for embedded systems designs and the most used modeling language in the embedded software industry [7, 407]. Furthermore, UML provides domain-specific extension mechanisms, and developers can access various MDD tools with UML support. For the approach presented in this thesis, IBM Rhapsody has been selected as an MDD tool for modeling and simulating software application models and the prototypical implementation of the concepts introduced in Section 6 (p. 139 ff.). In MARTE (cf. Section 2.6, p. 41 ff.), power consumption is not specified with the required granularity, and concepts to model dynamic power-related behavior are missing. With the PAP, this thesis introduces a UML profile to fill these gaps providing novel concepts to model the dynamic power-related behavior of an embedded system.

Related work presented in Section 2.8 (p. 65 ff.) discusses power consumption modeling at different levels of abstraction. Considering MDD, low-level and source code-based approaches (cf. [79, 154, 217, 344, 352, 370, 387]) are unsuitable for a power consumption estimation in early design phases due to the need for hardware-specific assembly code or source code in low-level languages such as C, which is automatically generated at later stages of the MDD process. Moreover, these approaches are limited to single components, typically the MCU. In further model-based approaches, e.g., based on mathematical models [55, 239, 262, 372, 426], workflow models [427], or AADL models [94, 353], aspects of the software application are either not considered at all or are modeled as abstract parts, e.g., as thread models with NFPs, such as minimum and maximum execution times. Such approaches are less suitable for analyzing the application or business logic of software in MDD. When simulated, the dynamic behavior is determined at runtime and depends on inputs defined by test cases. Work on power consumption modeling based on UML, SysML, or MARTE is more closely related to the approach presented in this thesis. However, existing related work is focused on estimating the power consumption of CPUs [3, 151], MCUs [23, 178, 199], SoCs [389], or even lower-level components such as clock generators [27]. For the simulation, UML class diagrams and UML sequence diagrams defining the software application are transformed into low-level and code-based approaches [199, 389] or represented as task models with expected characteristics [3, 151, 178]. Moreover, the dynamic behavior of the embedded system is limited to a set of predefined states for each hardware component associated with the operating modes of the software application [22, 23]. None of the concepts consider hardware models as an integral part of software applications for building system models. Since presented concepts for hardware models lack standardized descriptions for power- and energy-related NFPs, the impact of software applications on the overall power consumption is rarely considered.

Estimating the power consumption of software applications requires a software model with at least some application and business logic and models of all hardware components controlled or affected by the software application. A general modeling concept for hardware components is required for such a system view, which can model functional and non-functional aspects, reflect dynamic power-related behavior, and interact with software models. With the concept of hardware component models presented in Section 5.2 (p. 117 ff.), these gaps are addressed.

RQ4 – Early Evaluation of Energy-aware Software Applications in MDD

RQ4 aims to determine how the energy-related impact of software application models can be estimated and energy-related misbehavior detected when the hardware platform is not or is only partially available. To address RQ4, a novel system-wide approach is required to enable a power consumption estimation in early development phases based on virtual and physical hardware platforms.

Due to the cause-effect relationship between software and hardware, a grey-box approach is required causing the need for instrumentation and probing [367] for the power analysis to observe the internal state of the SUT during testing (cf. Section 2.7.5, p. 61 ff.). However, different challenges exist, especially for hardware instrumentation:

- Depending on the current state of each hardware component, the SUT may operate in the mA or µA range. A highly accurate measurement of such dynamics requires the use of different probes or advanced and expensive measuring devices.

- If a high level of granularity is required, each hardware component of the SUT has to be measured individually during execution which raises the need for multiple measuring devices while data has to be synchronized for the analysis.

To overcome this challenge, the complete SUT is considered as a single unit, accepting a loss of precision in the measurement. To still obtain the current state of each hardware component, simulation logs have to be generated and analyzed. Another topic to consider is the selection of proper MDD tools. While most of the MDD tools can simulate UML models, they lack support in analyzing NFPs, especially related to efficiency and performance attributes such as power and energy consumption [15]. Due to this, it is necessary to define a concept in which an external analysis tool can be coupled with the simulation environment of an MDD tool.

If the impact of the software application is taken into account early in MDD, e.g., by using simulation environments instead of source code execution, power-optimization measures are expected to be more significant and faster to implement (cf. Section 2.8, p. 65 ff.). Following the V-model development process used in MDD (cf. Figure 2.21, p. 59), MiL is the first method used as a quality gate to test and evaluate the software application model design. On the other hand, HiL, the latest in-the-loop testing paradigm, is used to identify issues with hardware components and hardware-software interactions. If NFRs are not fulfilled in HiL tests, especially if the problem is software-related (e.g., architecture, algorithms, or control of hardware components), all stages of the V-model have to be passed again before the evaluation can be repeated. To avoid an increased number of rework phases and higher costs to fix detected power-related issues [153, 361, 367], the evaluation of power- and energy-related NFRs should already be a testing goal in early development phases, as discussed in the problem statement (cf. Section 1.2, p. 4 ff.). Approaches such as rapid prototyping can be used to overcome the additional development and planning effort of the V-model development process when considering power- and energy-related NFRs. Rapid prototyping comprises techniques for rapidly implementing software applications and embedded systems as models and prototypes for early and fast testing [298]. The software application, which can be executable code or a simulated model [59, 240, 305], is evaluated in a real-world environment or a close equivalent [59] by using a fully-functional embedded system as a testbed in the later field of application.

However, the related work does not address the integration of a physical embedded system for early non-functional testing. In [161], hardware-specific source code is executed directly on the MCU, which is part of later MDD phases. Besides the MCU, other components of the embedded system are not considered. A middleware for communication with virtual and physical embedded systems is used in [61, 365, 366]. However, the software application model is not able to directly interact with embedded system components, e.g., sensors and actuators, which are only integrated as data sources on a functional level. Co-simulation [39, 209, 346] focuses on coupling simulation environments and models executed on different hosts, e.g., as a possibility to integrate physical hardware platforms into XiL testing environments. Protocols such as the DCP [254] are used for interoperability between simulated models, but they are not suitable for resource-constrained embedded systems.

Source code level approaches (e.g., [161]) are performed later in MDD after model transformations, followed by hardware-specific manual adjustment, compiling, and flashing steps. In [3, 22, 23, 151, 178], simulations are used as a virtual hardware representation of the MCUs to estimate power consumption. Other peripheral devices are not considered, which constitute a significant part of the system's power consumption [396, 429].

For an early estimation, e.g., when using MiL, related work has shown that no suitable approach exists to couple and integrate hardware behavior into the simulation environment so that the energy-related impact of software models can be estimated in early development phases of MDD. Finally, to the best of our knowledge, with discussed related work, software developers in MDD can hardly quantify the power-related impact of software application models. This also includes the detection of energy bugs located at the application level. To fill these gaps, this thesis provides novel concepts for a power consumption estimation in early development phases based on virtual and physical hardware without needing *edit-cross-compile-flash-debug* cycles [30, 393] for each test run. Furthermore, concepts for a low-level interaction between the software application model and the embedded system are presented. The analysis of the related work has pointed out the need to address energy-related aspects in software engineering. Considering design patterns for structuring software applications, no framework exists to document best practices focusing on energy and power properties. Additionally, there are still open questions for software applications in MDD regarding the system-wide modeling and evaluation of power- and energy-related properties in early development phases. The following chapters introduce solutions to address RQ1 to RQ4.

Chapter 3

Overview

This chapter provides an overview of the overall approach of this thesis and introduces fundamental concepts used in modeling and estimating the power consumption of software applications as carried out in this thesis. Section 3.1 introduces a workflow from a developer's perspective as an illustration of the synergy and combined use of the main contributions described in-depth in Chapter 4 (p. 89 ff.) and Chapter 5 (p. 115 ff.). Furthermore, concepts that provide fundamental definitions for various parts of the overall approach are presented. Section 3.2 presents the concept of *scenarios*, which describe a set of constraints, requirements, and properties of the system and environment that are valid during a test case execution. In Section 3.3, introduces *energy bugs* as a description of the non-functional misbehavior of a system. Scenarios and energy bugs are two closely related concepts. A violation of energy-related NFRs defined within a given scenario may indicate the presence of energy bugs. Thus, a scenario describes the boundaries between normal behavior and misbehavior (energy bug) of the system and allows the power estimation process (cf. Chapter 5, p. 115 ff.) to decide whether a prediction is within or outside the acceptable range.

3.1 Developer Workflow

This section presents the developer workflow as an overview to demonstrate how the proposed concepts complement and interact with each other and to illustrate their application in different phases of the MDD process. The workflow is illustrated as a UML activity diagram in Figure 3.1 and consists of a set of roles, twelve actions and sub-actions, and six artifacts. To categorize activities based on responsibilities, three roles, namely requirements engineer, software engineer, and test engineer, have been defined and expressed as swimlanes. Note that these roles are more illustrative and not final in any sense.

The first action of the developer workflow shown in Figure 3.1 is performed during requirements analysis. In sub-action 1(a), functional and non-functional requirements and scenarios (cf. Section 3.2, p. 81 ff.) are defined. Scenarios describe a set of conditions and constraints to specify the actual environment in which the software application is evaluated. In addition, scenarios also specify configuration parameters and parameters for hardware components that are valid during the execution of test cases. NFRs are based on the formal description proposed in Section 3.3 (p. 84 ff.) to address RQ1. The result of action 1(a) is a list of scenarios and requirements (artifact A1). The test engineer may use the list to set up the test environment in later steps. It also defines the basis for detecting energy bugs of the SUT. In action 1(b),

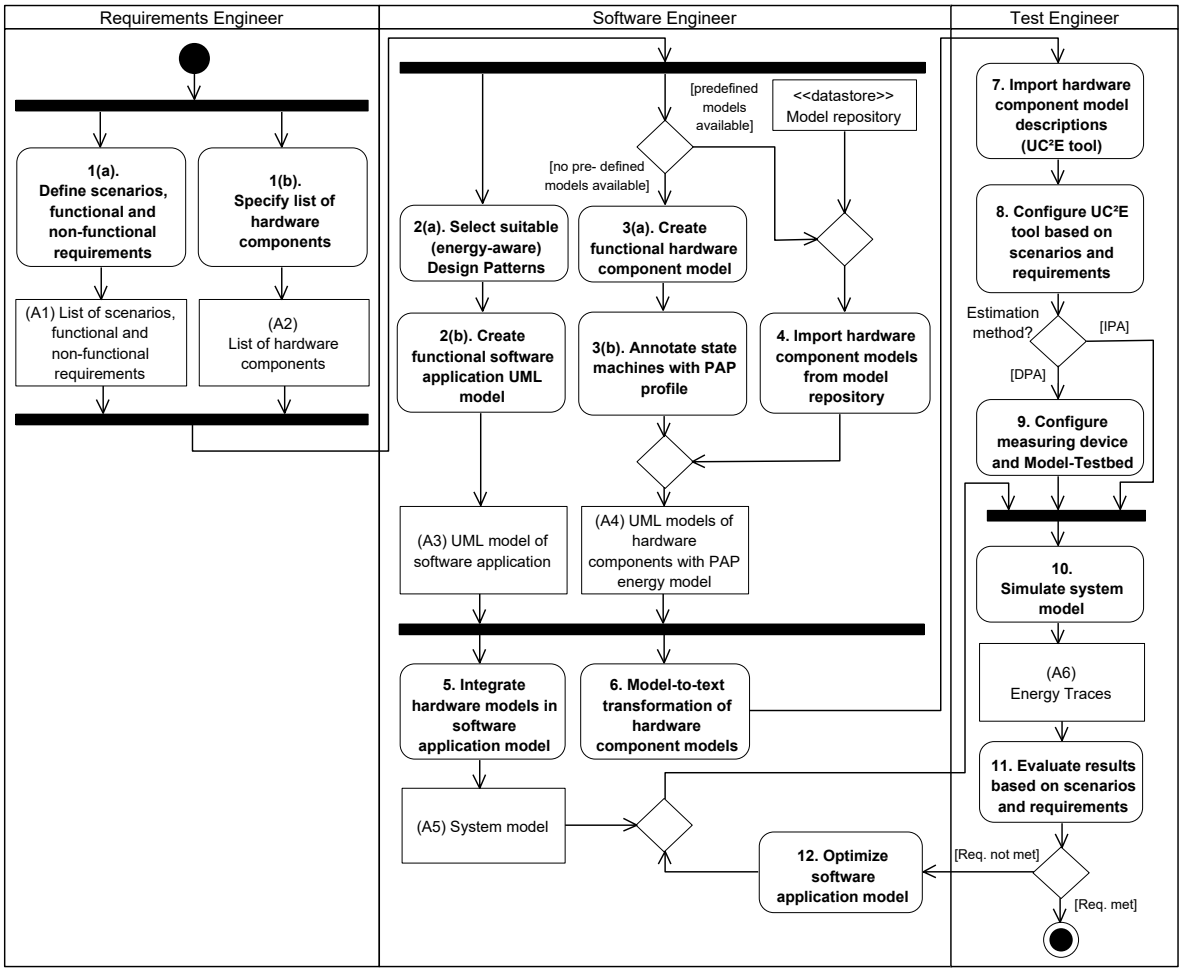


Figure 3.1: Developer workflow describing the usage of the presented concepts (UML 2.5 activity diagram notation).

the requirements engineer creates a list of hardware components (artifact A2), e.g., MCUs and peripheral devices such as sensors and actuators. The hardware components may be selected to fulfill requirements or specifications from other divisions omitted in Figure 3.1, such as the hardware engineering or costing department.

The resulting artifacts A1 and A2 also serve as input for the software engineer who defines a functional UML model of the software application (artifact A3) in actions 2(a)–2(b). Addressing RQ2, energy-aware software design patterns introduced in Chapter 4 (p. 89 ff.) may be used during the initial design phase of the software application model in action 2(a) as structural and behavioral solutions to explicitly address energy-related aspects. However, applying energy-aware software design patterns from the design pattern catalog (cf. Section 4.4, p. 94 ff.) is an optional design choice that depends on the problem to be solved and the target system used. In parallel to the development of the software application model, hardware component models are defined in actions 3 to 4. If no predefined hardware component models exist, they have to be created by the software engineer in action 3(a) and annotated with the *Power Analysis Profile (PAP)* in action 3(b) to apply a set of stereotypes to model energy-

related NFPs resulting in an energy model containing energy-, power-, and time-related aspects. The process of creating hardware component models and the definition and usage of the PAP are part of the power estimation design concept for MDD described in-depth in Chapter 5 (p. 115 ff.). The resulting hardware component models with the PAP-based energy model (artifact 4) in Figure 3.1 represent abstractions and proxies of hardware components that can interact with the software application model. Both hardware component models and the UML-based PAP are contributions of this thesis to answer RQ3 and are introduced in Sections 5.2 (p. 117 ff.) and 5.3 (p. 124 ff.), respectively. If hardware component models already exist, for example, stored in a model repository of an MDD tool, they can be imported into the MDD project described as action 4. Note that the definition of a model repository, as shown in Figure 3.1, is only conceptual and outside the scope of the thesis. The model repository contains hardware models from existing projects or uses model transformations as mentioned in Section 2.5.2 (p. 40 f.) to derive hardware component models obtained by other approaches, e.g., as described in [127]. To achieve a system-wide view, the software application model (artifact A3) and hardware component models (artifact A4) are combined into a system model (artifact A5) as a result of action 5. The system model facilitates the analysis process by combining software and hardware aspects of the system into a single representation based on UML. For the exchange of model information between the MDD tool and the analysis tool, a model-to-text transformation of hardware component models is performed in action 6. As an interchange format, the lightweight and *JavaScript Object Notation (JSON)*-based transformation format [338] specified in Section 6.1.1 (p. 140 ff.) is used.

In action 7, the test engineer uses the resulting hardware component model descriptions as an import for the analysis tool as the first steps of the evaluation process. As proof-of-work to address RQ4, an exemplary analysis tool denoted as *Unit for Central Control and Estimation (UC^{2}E)* has been developed in this thesis, which is introduced in Section 6.4 (p. 155 ff.). It parses hardware component model descriptions and extracts information necessary to derive energy models of each hardware component for the estimation process. In action 8, the list of scenarios and requirements (artifact A1) is used to further configure the UC^{2}E tool, e.g., the test environment. Depending on the selected power analysis method (cf. Section 5.4, p. 135 ff.), the measuring device and the *Model-Testbed* (cf. Section 6.5, p. 161 ff.) have to be configured in action 9 before the simulation of the system model (artifact A5) is performed in action 10. As further contributions to address RQ4, extensions such as a HAL (cf. Section 6.3, p. 149 ff.) and protocols for data exchange (cf. Section 6.2, p. 145 ff.) are developed for the simulation of system models (artifact A5) to achieve a power consumption estimation in early design phases. By this, an extensive tracing can be achieved, which enables an online and offline estimation by the UC^{2}E tool. As a result of the simulation, energy traces (artifact A6) are obtained and used to evaluate NFRs and estimate the power consumption in action 11. The developer workflow is completed, if no power-related NFRs are violated. When violations exist, the software application model must be optimized by fixing detected energy bugs in action 12. Similar to the development of the software application model in action 2(a) to 2(b), energy-aware design patterns may also be used by software engineers for adjustments during the re-design phases in action 12. Afterward, the simulation and analysis process may be repeated, starting from action 10. Note that actions 5 to 6 may be executed automatically by the MDD tool, while extensive algorithms of the analysis tool may automate actions 9 and 11.

Figure 3.2 illustrates the vision of the enhanced MDD process with applied developer workflow for the model-based design and development of software applications. Following the V-model shown in Figure 2.21 (p. 59), the MDD process can be divided into five distinct phases. It starts with the *architecture* phase of the software application on the left side of Figure 3.2, followed by a *design*, *simulation*, and *integration* phase until reaching the *rollout and operational* phase of the final product, e.g., an IoT sensor node, on the right side of Figure 3.2. The x-axis shows the time spent on development, which increases from the left to the right. The time and cost required to analyze and correct defects increases with each completed MDD phase. The y-axis shows the consideration of power-related aspects and the accuracy of the power consumption estimation as two mutually dependent parameters.

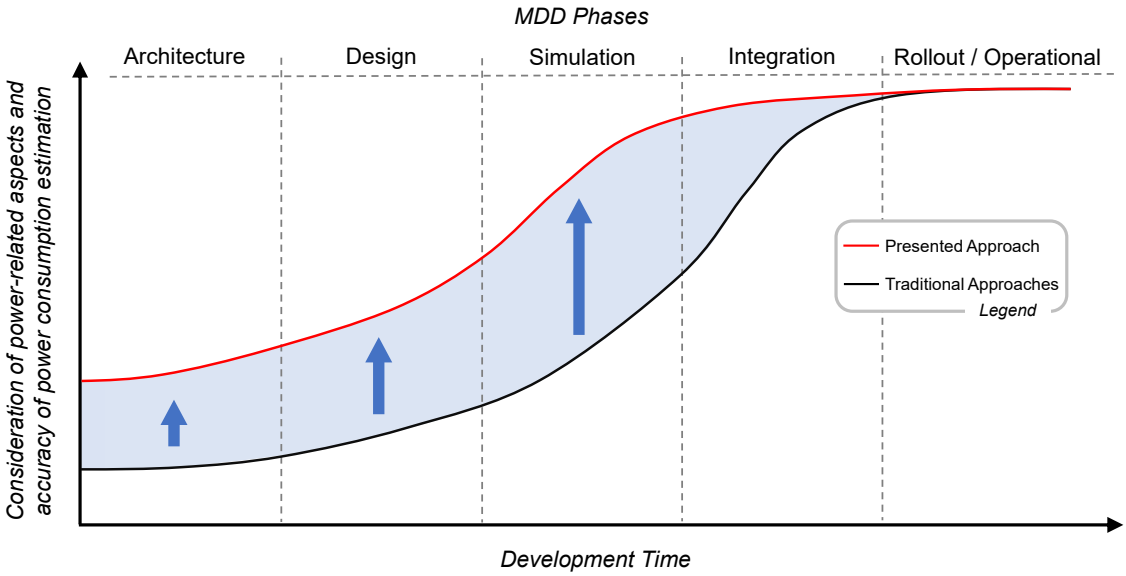


Figure 3.2: Vision of the enhanced MDD process with applied developer workflow.

By applying the developer workflow as depicted in Figure 3.1 (p. 78), power aspects may already be considered during the architecture and design phase. For example, energy-aware design patterns may be used in the architecture phase to define an energy-aware software application (cf. Chapter 4, p. 89 ff.). The software application model may be extended during the design phase with hardware component models (cf. Section 5.2, p. 117 ff). Compared to traditional approaches that do not consider energy-related aspects in early development phases, a static analysis based on the integrated energy models of hardware component models may be performed to evaluate, e.g., NFRs. However, the most considerable improvement may be achieved when applying the power consumption estimation approach during the simulation phase. Based on the power analysis methods presented in Section 5.4, the software application model is executed within a simulation that enables the evaluation of NFRs and the detection of energy bugs (cf. Section 3.3, p. 84 ff.). The approach presented in this thesis not only increases the accuracy of power analysis in early development phases but may also save time and reduce costs by avoiding time-consuming and cost-intensive redesign loops due to the early detection of energy-related misbehavior and violations of NFRs. In traditional approaches, those aspects are typically evaluated during the integration phase of the software application and hardware platform (cf. Section 2.7.4, p. 57 ff.).

3.2 Scenarios

Embedded systems, such as IoT smart objects introduced in Section 2.2 (p. 21 ff.), are considered reactive. Unlike transformative systems, reactive systems are characterized by a non-terminating and event-driven behavior continuously reacting to external and internal events [155]. External events typically arise through interaction with the environment. The system accepts inputs from the environment, e.g., by using sensors, and changes the internal state in response to the input event. Defining the expected reactive behavior while executing test cases is crucial when testing embedded systems. For instance, when measuring the power consumption of the *System Under Test (SUT)* during test case execution, developers need to know the context, e.g., what kind of behavior caused the measured current draw. For this, an essential element in the specification of tests is the need for clear definitions of aspects that may affect the SUT during execution.

The behavior of a software application generally depends on the environment and context in which it is executed. As a result, the context directly or indirectly influences power demand and consumption, which is not sufficiently considered during testing. For an improved test specification, the concept of scenarios tries to address this problem by defining a set of conditions and constraints that apply for a time period T under consideration. For example, some smoke or fire detectors are sensitive to the current ambient temperature, which affects their behavior and, thus, the expected power consumption. Poor network connectivity can result in multiple transmit or receive cycles increasing the power consumption of a wireless module.

In general, a scenario S covers the following aspects for the evaluation of a SUT:

- Conditions and properties of the environment affecting the functional or non-functional behavior of the SUT.
- Initial values for parameters of the system or individual components with an influence on the power-related behavior. These parameters are either unaffected or cannot be addressed by the software application.
- Power and energy characteristics to specify the expected energy-related behavior and limits.

Considering the introduced aspects, a scenario S may be expressed as a list of conditions, constraints, and requirements. From a test case perspective, a scenario requires specific parameters to be configured and constraints to be met, allowing them to be also interpreted as requirements for a given scenario. Since a requirement is typically described using a unique identifier, each group of requirements and properties can be expressed as a vector, resulting in Definition 3.1.

Definition 3.1 *We define a scenario* $S = \{\vec{R}_g, \vec{R}_e, \vec{R}_p, \vec{R}_{en}\}$ *as a list of requirements and conditions valid for the time period in which the scenario* S *is active within a test case. The elements of a scenario* S *are:*

$\vec{R}_g = (r_{g_1}, \ldots, r_{g_n})$*, as a vector of general requirements,*

$\vec{R}_e = (r_{e_1}, \ldots, r_{e_n})$*, as a vector of environmental requirements,*

$\vec{R}_p = (r_{p_1}, \ldots, r_{p_n})$*, as a vector of (predefined) system- and component-specific properties,*

$\vec{R}_{en} = (r_{en_1}, \ldots, r_{en_n})$*, as energy-related requirements (cf. Definition 3.2, p. 85).*

The description of scenarios is not limited to a specific form of expression. The requirements diagram introduced in the SysML specification [279] is particularly suitable for describing and representing requirements in MDD. In SysML, requirements are expressed from a system perspective with stereotypes available for the description of behavior in terms of functional requirements («functionalRequirement»), physical constraints («physicalRequirement»), or performance-related requirements («performanceRequirement») [279]. However, the SysML specification lacks stereotypes to specify energy-related and environmental-related requirements and does not consider parameter definitions that may affect functional and non-functional behavior. The extension mechanism of UML has been used to design a set of new stereotypes for SysML, as shown in Figure 3.3. With the newly defined stereotypes

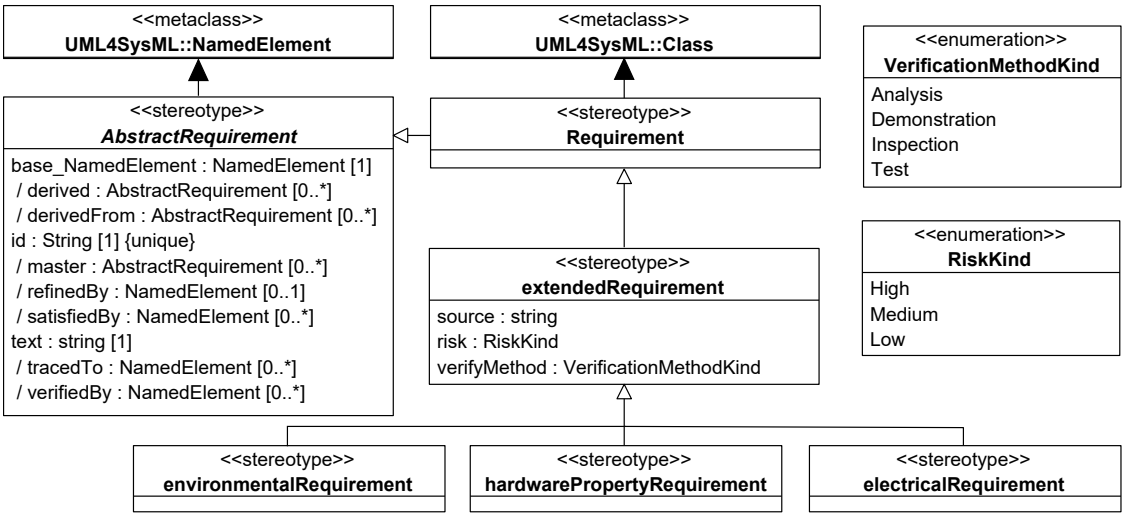


Figure 3.3: Stereotypes to describe requirements for scenarios (UML 2.5.1 profile diagram).

«environmentalRequirement»,«hardwarePropertyRequirement», and «electricalRequirement», the requirements of a scenario can be described and grouped logically. An excerpt of a scenario definition with different requirements utilizing the newly introduced stereotypes is pictured in Figure 3.4. For the exchange of scenarios between MDD tools, interchange formats such as the *Requirements Interchange Format (ReqIF)* [274] may be used.

In the exemplary scenario definition for a generic environmental sensor shown in Figure 3.4, the atmospheric pressure and ambient temperature, as environmental requirements, may vary between 1000–1005 mbar and 12–35 °C while the scenario is active during test case execution. As predefined hardware properties, the oversampling rate of the sensor is defined by the requirement *ReqS1.2.1.2*. Since such properties may affect the behavior of the environmental sensor, their specification is necessary for the power consumption estimation process. The requirements annotated with the «electricalRequirement» stereotype may be used to specify an energy-related NFR, as explained in Section 3.3 (p. 84 ff.). A single scenario S_{tc} may be defined for the test case tc if the defined requirements are considered constant. However, defining a list of scenarios for a single test case is also possible. Imagine a test case where the SUT should send a notification message whenever the ambient temperature reaches a threshold value of, e.g., 20 °C, for which two distinct scenarios may be defined. The first scenario may specify an ambient temperature below 20 °C, while for the second scenario, the requirement for the ambient temperature is set to a value equal to or greater than 20 °C while

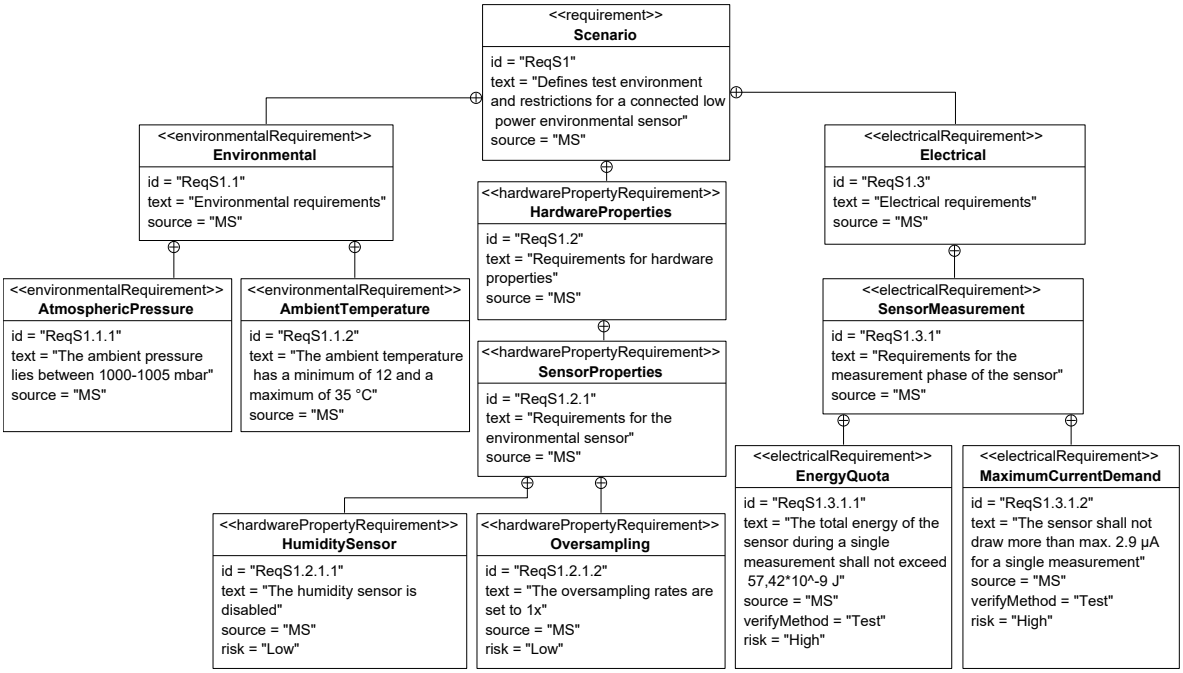


Figure 3.4: Excerpt of a scenario definition with applied scenario-specific stereotypes (SysML 1.6 requirement diagram notation).

leaving other requirements unchanged. Both scenarios may be defined as sub-scenarios for the overall scenario S_{tc} of test case tc with different thresholds for energy-related behavior. To specify the order of sub-scenarios for a test case execution with a time period T, a notation is introduced in Equation (3.1) to define a sequence of scenarios S_i along with points in time $t_i \in T$ as their end of execution:

$$S_{tc} = [\langle S_0, t_0\rangle, \langle S_1, t_1\rangle, \ldots, \langle S_{n-1}, t_{n-1}\rangle, \langle S_n, t_n\rangle](t) \begin{cases} S_0 & 0 \leq t < t_0 \\ S_1 & t_0 \leq t < t_1 \\ \vdots & \\ S_{n-1} & t_{n-2} \leq t < t_{n-1} \\ S_n & t_{n-1} \leq t \leq t_n \end{cases} \tag{3.1}$$

It is also possible to define the scenarios of a test case as periodic, denoted as $\overline{S}_{tc}$, which can be written as:

$$\overline{S}_{tc} = [\langle S_0, (i \cdot T) + t_0\rangle, \langle S_1, (i \cdot T) + t_1\rangle, \ldots, \langle S_{n-1}, (i \cdot T) + t_{n-1}\rangle, \langle S_n, (i \cdot T) + t_n\rangle](t) \tag{3.2}$$

where $i \geq 0$ defines the number of iterations in zero-based numbering. For instance, if $i = 1$, each scenario is active for two time periods within a test case executed for a total execution time of $2 \cdot T$.

Scenarios may also be extended with probabilities to describe more realistic test cases, especially for the event-driven domain of embedded systems. A scenario with sub-scenarios containing probabilities is denoted as $\widehat{S}_{tc}$ and can be defined similarly to Equation (3.1), for

instance:

$$\widehat{S}_{tc} = [\langle S_0, P_{t_0 S_0}, t_1\rangle, \langle S_1, P_{t_1 S_1}, t_2\rangle, \langle S_2, P_{t_2 S_2}, t_3\rangle, \langle S_3, P_{t_3 S_3}, t_4\rangle, \langle S_4, P_{t_3 S_4}, t_5\rangle](t) \tag{3.3}$$

with

$$\sum_{i=0}^{n} P_{t_m S_i} = 1, \text{ for each } t \in T. \tag{3.4}$$

Equation (3.4) describes the probability of S_i being active at a specific time t_m. In Equations (3.1) to (3.2), scenarios are executed sequentially. By introducing probabilities, one of several possible scenarios $S_n \in \widehat{S}_{tc}$ will be active at a given time $t \in T$. Therefore, the tuple definition is extended with an additional parameter to define the probability of a scenario being executed at a specific point in time. For instance, one may consider the last two scenarios of $\widehat{S}_{tc}$ in Equation (3.3), namely $\langle S_3, P_{t_3 S_3}, t_4\rangle$ and $\langle S_4, P_{t_3 S_4}, t_5\rangle$. Both have a probability of being active at t_3 which defines the point in time where the execution of S_2 ends. If S_3 is selected, the scenario will be active from t_3 to t_4. Otherwise, S_4 will be active from t_3 to t_5. Developers may also be able to graphically model scenarios of a test case, e.g., as defined in Equation (3.3), using UML state machine diagrams (cf. Appendix A.3, p. 266 ff.). With such a model-based representation, scenarios are suitable to be used in MBT.

3.3 Energy Bugs

To describe the behavior of a SUT consuming more energy than required to fulfill the intended task, the term *energy bug* has been introduced as an umbrella term to describe a broad range of bugs associated with energy consumption. Ideally, energy bugs should be addressed in early development stages, for instance, during the software architecture or component design phase, as illustrated by the V-model in Figure 2.21 (p. 59) and discussed in Section 2.9 (p. 72 ff.), using accurate simulations. By this, time and resources, e.g., financial and human resources, can be saved compared to an evaluation of non-functional aspects later in the development cycle resulting in multiple re-design phases. As previously stated in Section 2.3.3 (p. 30 ff.) and Section 2.8 (p. 65 ff.), researchers have published numerous works to analyze energy consumption (cf. Figure 2.24, p. 66) and to detect and describe energy bugs. Since energy bugs in previous work mainly refer to the effects, but neither a clear definition nor specific sources of energy bugs have been provided, a revision of the energy bug definition is required.

An initial version of the definition and classification of energy bugs has been published in [341] and revised in this thesis due to new findings. The section provides definitions for the terms energy bug and energy-aware systems in Section 3.3.1, as well as a classification scheme in Section 3.3.2. Moreover, a basic example to illustrate the relationship between energy bugs, scenarios (cf. Section 3.2, p. 81 ff.), and NFRs (cf. Section 2.3, p. 27 ff.) is presented in Section 3.3.3.

3.3.1 Energy Misbehavior

In the context of this thesis, the term energy bug refers to an unintended behavior of an embedded system causing a higher power consumption, which is unnecessary to provide the current functionality. Generally, energy bugs refer to the behavior of a SUT, defined as a hardware platform with a finite set of n independent hardware components C, so that:

$$SUT = \{C_1, C_2, ..., C_n\}$$

Each hardware component C_i has its own energy-related behavior. The power level of the SUT at a specific time is defined by the active state of each hardware component $C_i \in SUT$. When considering Equations (2.1) and (2.7) of Section 2.1.1 (p. 17 ff.), the electric power consumption P_{C_i} for a specific time $t \in T$ and the energy consumption E_{C_i} for a time period T of a hardware component C_i can be expressed as:

$$P_{C_i}(t) = U_{C_i}(t) \cdot I_{C_i}(t) \tag{3.5}$$

$$E_{C_i}(T) = \int_{t=0}^{T} P_{C_i}(t)\, dt \tag{3.6}$$

Note that the definition of the SUT depends on the test case and may consist of the complete system or a set of hardware components required for the test case execution, e.g., an MCU, a set of sensors, or a radio module. Considering previous calculations, the electric power P and energy E of a system consisting of n hardware components may be defined as:

$$P_S(t) = \sum_{i=1}^{n} P_{C_i}(t) \tag{3.7}$$

$$E_S(T) = \int_{t=0}^{T} P_S(t)\, dt = \int_{t=0}^{T} \left(\sum_{i=1}^{n} P_{C_i}(t) \right) \tag{3.8}$$

A new definition of energy bugs is introduced to address RQ1, incorporating previous considerations presented in Section 2.3.3 (p. 30 f.) and considering new aspects. A tuple of two parameters $\langle E_{qu}, I_{dmax} \rangle$ is specified to describe the energy-realted behavior of a SUT. The energy quota E_{qu} describes the energy available for a period T. The second parameter I_{dmax} specifies the maximum current demand for a point in time $t \in T$. An energy bug infringes at least one of the two conditions and, thus, describes a deviation from a previously defined NFR. If no NFRs are violated, the SUT can be considered energy bug-free.

Definition 3.2 *We define a SUT (or a subset of components C_i) to be* ***energy bug-free*** *if the conditions in Equations (3.9) to (3.10), as part of a single NFR or scenario S, are satisfied:*

$$E_S(T) \leq E_{qu} \tag{3.9}$$

$$\max(I_S(t)) \leq I_{dmax} \tag{3.10}$$

When an energy bug occurs during test case execution, at least one of the rules presented in Definition 3.2 is violated. The definition of energy bugs is not limited to specific elements, can be phrased on multiple levels, and can be hardware and software related. Hardware-related definitions may be defined for the complete system or the behavior of single components. On the other hand, software-related definitions may refer to single functions of the software application or a program flow, e.g., expressed as a UML sequence diagram [275].

In addition, non-functional misbehavior of a SUT and unused energy-saving opportunities have to be considered. For this, the notion of an energy-aware or energy-efficient SUT is introduced, which relates to the parameters for defining energy bugs in Definition 3.2.

Definition 3.3 *We define a SUT as* ***energy-aware*** *or* ***energy-efficient*** *if we apply a set of measures that minimize $E_S(t)$ and/or $max(I_S(t))$.*

In general, the measures are not subject to any restrictions as long as they positively affect one or both parameters. It is important to note that they should not negatively impact the system's

functional behavior but may still influence other NFRs such as security, real-time behavior, and response times as a side effect. Examples of measures are energy-aware software design patterns as part of the design pattern catalog introduced in Chapter 4 (p. 89 ff.). The provided framework to describe energy-aware software design patterns aims to minimize the parameters mentioned in Definition 3.3 by adapting the architecture and behavior of software applications. Other measures not related to design patterns may include compiler optimizations, which are not in the scope of this thesis.

3.3.2 Classification

As elaborated in a previous publication [341], energy bugs can arise from various sources. The first distinction can be made between hardware-related and software-related energy bugs. With subcategories for both sources to aggregate groups of energy bugs with similar causes, the following classification is introduced:

- Hardware-related Energy Bugs
  - *Type A*: An incorrect hardware design or a faulty component leading to unexpected power consumption during runtime.
  - *Type B*: Unknown or unconsidered consumers, primarily hardware components, leading to unexpected power demand.
- Software-related Energy Bugs
  - *Type C*: Flaws in the software design, inappropriate design patterns, or incorrect hardware usage causing increased power consumption.
  - *Type D*: Software preventing hardware components (e.g., MCUs, peripheral devices) from entering low-power modes, increasing the system's power consumption.

The presented classification provides a more general categorization of energy bugs in the embedded systems domain, which goes beyond the scope of mobile devices described in Section 2.3.3 (p. 30 ff.). The presented approach introduced in Chapter 5 (p. 115 ff.) is able to detect both sources of energy bugs, as the case study in Chapter 7 (p. 175 ff.) demonstrates.

The energy bug types *A* and *B* are located on the hardware layer of the SUT. Type *A* energy bugs may be caused by the hardware layout and become visible when environmental conditions change. An example of Type *B* energy bugs may be additional *Light-emitting Diodes (LEDs)* in the layout to indicate the system's status, leading to unexpected power demand. Such components can easily be overlooked when predicting the energy consumption of a SUT. Troubleshooting may require extensive hardware analysis using additional equipment such as multimeters, oscilloscopes, or logic analyzers. In the worst case, it may be necessary to replace faulty hardware components or redesign the underlying hardware layout.

Type *C* and *D* energy bugs are related to the software layer of the embedded system. Type *C* energy bugs may arise from software design flaws, unsuitable design patterns, a faulty or unoptimized configuration, and unnecessary hardware-software interactions. Furthermore, an incorrect or non-optimized utilization strategy of system resources, e.g., peripheral devices, can lead to a higher power consumption and Type *C* energy bugs [335]. For instance, peripheral devices can be locked into a higher power mode if they are enabled for a specific part of the software workflow and not disabled after the workflow has been finished. Furthermore, they

often have a fixed energy offset called the energy tail [32, 34] whenever certain operations are executed. If these operations are executed at a high frequency, the offset significantly affects the overall consumption of the system. In addition, unnecessarily high sampling rates for sensors may lead to avoidable extra power consumption. On a lower level, flaws in the software design, e.g., blocking method calls, may prevent peripheral devices or the complete system from entering a lower power state. Additionally, unoptimized source code (e.g., avoidable wait cycles) may lead to more extended execution times while keeping the MCU busy or active for a longer time frame. Programming errors such as unreachable code sections may prevent hardware accesses (e.g., configuration or power state changes) from being executed. Moreover, the inaccurate use of *Application Programming Interfaces (APIs)* and drivers (e.g., due to inexperienced software developers or exotic hardware components) affects the power consumption of a peripheral device.

In contrast, Type *D* energy bugs are related to the influence of software applications on the power state of peripheral devices. Hardware components of an embedded system realize different low-power modes, also known as sleep modes, that may be active if the component is not in use. Unintentional accesses can prevent these components from switching into sleep mode (no-sleep bug [295]). They may also be bound to a part of the software application too early, too long, or both, causing them to operate at high power consumption for longer than necessary. An example of such an energy bug is described in [391], where the bootloader of the MCU, as part of its default behavior, generates debug messages on the UART interface as soon as the MCU is switched from an active into a low-power mode. Imagine a device already powered down by the software application. Messages on a communication bus, e.g., UART, may be interpreted falsely as a wake-up signal so that the peripheral device sets itself into a higher power mode, e.g., idle, without notifying the software layer. Because hardware components are not operating in their expected power mode, the power consumption of the system deviates from the expected level.

3.3.3 Example

This section provides a basic example for the appliance of the tuple definitions (cf. Definition 3.2, p. 85) to demonstrate the relation between scenarios, requirements, and energy bugs. This example refers to the IoT sensor node example presented in [341] and used for the case study as part of the Evaluation in Chapter 7 (p. 175 ff.). However, a detailed understanding of the case study is not necessary at this point. Imagine an IoT sensor node as SUT, which uses LPWAN as a wireless communication technology to transfer measurement values to a server. The behavior of the SUT can be divided into different phases, while a scenario can be defined for each of these phases if the entire behavior is evaluated within a test case, for instance:

$$S_{tc} = [\langle S_{measure}, t_{measure}\rangle, \langle S_{transmit}, t_{transmit}\rangle, \langle S_{sleep}, t_{sleep}\rangle] \tag{3.11}$$

Exemplary for the scenario $S_{transmit}$, the LPWAN module might be the most critical component to be considered. A requirement engineer may specify the following two NFRs to define limits considering the energy consumption and the peak demand during a single transmission to avoid excessive load on the energy source.

The total energy of the LPWAN module during a single transmission shall not exceed $42.2 \cdot 10^{-3}$ *J.*

The LPWAN module shall not consume more than 64 mA when transmitting messages.

For the evaluation, further requirements according to Definition 3.1 (p. 81) have to be specified, e.g., of the environment and properties of the LPWAN module. Figure 3.5 shows an excerpt of the defined scenario $S_{transmit}$ as SysML 1.6 requirement diagram. The scenario not only

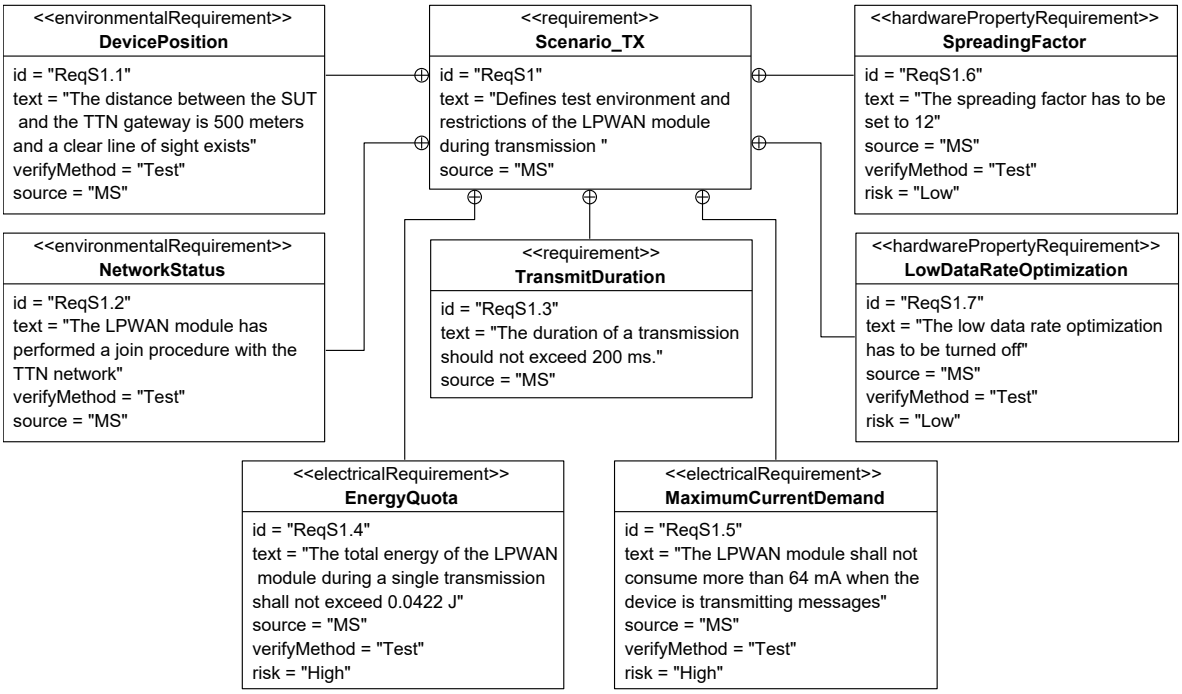


Figure 3.5: Requirements for a LPWAN module of an exemplary scenario (SysML 1.6 requirement diagram notation).

specifies the expected energy-related behavior of the SUT but also addresses aspects of the environment and the intended configuration of the LPWAN module as strong influencing factors on the non-functional behavior. Due to the definition requirements *ReqS1.4* and *ReqS1.5* in Figure 3.5, the energy consumption for a single transmission as the upper threshold and the maximum electric current consumption during the transmission can be considered. Both aspects are affected by the size of the data to be sent and the currently active configuration of the LPWAN module. An energy bug exists if at least one of the two introduced NFRs is violated. The limits of an energy bug-free system for the scenario $S_{transmit}$ can be defined as $\langle(42.24, mJ), (64, mA)\rangle$. In addition, the energy quota E_{qu} and the maximum current demand I_{dmax} may be used to express and quantify energy bugs themselves.

This chapter introduced the closely related concepts of scenarios as a structured collection of NFRs and energy bugs as violations of energy-related requirements. Both are expressed as a tuple with the energy quota and the maximum current demand. These two concepts form the basis for the energy-aware software design patterns in Chapter 4 (p. 89 ff.) and the modeling and estimation of power consumption in Chapter 5 (p. 115 ff.).

Chapter 4

Software Design Pattern Framework

This section provides a framework to derive and describe best practices for the design of energy-aware systems (cf. Definition 3.3, p. 85). Energy-aware design patterns may be used to initially design or optimize software applications after executing test cases and evaluating energy- and power-related NFRs. While the description of design patterns as a process must be completed in advance, their application is part of actions 2(a) and 12 in the proposed developer workflow shown in Figure 3.1 (p. 78). The following sections provide a set of contributions to overcome RQ2 for the design of energy-aware systems. Section 4.1 outlines the intended goal of the developed software design pattern framework. In Section 4.2, the process of identifying design patterns is briefly summarizes, while in Section 4.3, a novel template for the uniform description of software design patterns is presented. A first catalog of energy-aware software design patterns is introduced in Section 4.4.

Initial versions of the framework and the design pattern catalog have been published in [337] and revised and extended in follow-up work [338]. Selected novel energy-aware design patterns of the design pattern catalog have been published separately in [392].

4.1 Introduction

Software design patterns generally describe generic and programming language-independent *"generalized solution[s] to a commonly occurring problem."* [99]. In Section 2.4.1 (p. 32 ff.), different approaches for a systematic and abstract description of such software design patterns have been discussed. The main drawback of design pattern templates presented in Section 2.4.1 (p. 32 ff.) and scientific research mentioned in Section 2.4.2 (p. 34 ff.) is the lack of including the impact and side effects on power- and energy-related NFPs as the subject of NFRs in the description of software design patterns. Additionally, metrics to estimate the impact of a design pattern on NFRs, e.g., related to power and energy, when applied to a software application, are not part of the proposed templates.

This chapter provides a novel solution to describe energy-aware software design patterns, explicitly addressing energy-related aspects of the software application while interacting with hardware resources. The proposed framework is intended to provide the following elements:

- A uniform template to describe the key elements of energy-aware design patterns regardless of programming languages and peripheral devices.

- A section to describe the impact on power and energy using a set of quantifiable values (metrics). By this, the gap between the hardware and software layers can be addressed, and the cause-effect relationship between software and hardware can be considered.
- A uniform graphical description of the behavior resulting from applying a design pattern w.r.t. power consumption and time behavior aspects.

The design pattern framework provides a template to describe reoccurring solutions for the design of software applications in an abstract manner which may help developers to understand the problem area more in-depth and provide a kind of guideline for the development phase. Moreover, with the proposed metrics, a direct comparison between different design patterns becomes possible, which supports developers in selecting the most appropriate design pattern without the need to implement and evaluate each design pattern beforehand. Due to different characteristics, it should be noted that not every design pattern is suitable for every type of hardware. Additionally, there might exist tradeoffs between functional and non-functional requirements associated with the use of design patterns. Optimizing the software application in terms of power consumption, for example, may have a negative impact on the time behavior, which developers must carefully monitor [291].

Before the adapted design pattern template and the design pattern catalog are introduced in Sections 4.3–4.4, the process for identifying energy-aware design patterns is briefly described in Section 4.2.

4.2 Design Pattern Identification Process

This section describes the method and sources used to discover energy-aware design patterns provided by the design pattern catalog in Section 4.4 (p. 94 ff). Identifying design patterns and adding them to a catalog is often referred to as pattern mining [98, 99]. The process is naturally associated with discovering new design patterns rather than their invention [124], mainly because one has to notice that the solution covered by the design pattern in some context is similar to a solution in another context. However, there is no uniform process for the creation of design patterns. Instead, several guidelines for writing design patterns exist [248, 414]. An approach similar to the one described in this section has been presented in [120] as an iterative process containing the three phases of *pattern identification*, *pattern authoring*, and *pattern application* with multiple sub-activities for each phase.

For the pattern identification process carried out in this thesis, the research for energy-aware design patterns has not been limited to specific domains such as automotive or IoT. The following sources have been elaborated and considered as part of the identification process:

- *Literature* to identify existing software design patterns, which may also impact power- and energy-related NFPs.
- *Technical documentation* and *datasheets* published by manufacturers for their product lines intended for embedded system developers.
- *Whitepapers* of manufacturers dealing with the ideal use of their products, e.g., sensors and actuators, or explain technologies and domains in which their products are intended to be used.
- *Scientific publications* with a focus on software design patterns and embedded systems.

- *Own experience* based on findings from lectures, seminars, and research projects, e.g., [286], carried out in cooperation with small and medium-sized enterprises.

If a problem solution described in at least one of the above sources has been identified as essential and generalizable (pattern identification), the solution approach has been transformed into a new energy-aware design pattern (pattern authoring) using the proposed design pattern template presented in Section 4.3 (p. 91 ff.). Afterward, the energy-aware design pattern was added to the catalog presented in Section 4.4 (p. 94 ff.).

Most energy-aware design patterns of the design pattern catalog have gone through the first two phases since they are based on existing principles whose effectiveness has already been proven. However, newly discovered energy-aware design patterns based on own research, e.g., as published in [392], have also gone through the pattern application phase [120] to validate the described solutions.

4.3 Adapted Design Pattern Template

This section introduces the novel design pattern template for energy-aware software design patterns. The notation of the pattern template is based on work and concepts presented in Section 2.4 (p. 32 ff.). However, it has been modified to consider energy and timing aspects. The template also provides a set of metrics to quantify the impact of the design pattern and a new graphical representation to illustrate the behavior in an abstract form. An overview of the design pattern template is shown in Figure 4.1. The structure is divided into three main elements: *General Information*, *Description*, and *Impact on Non-functional Requirements*.

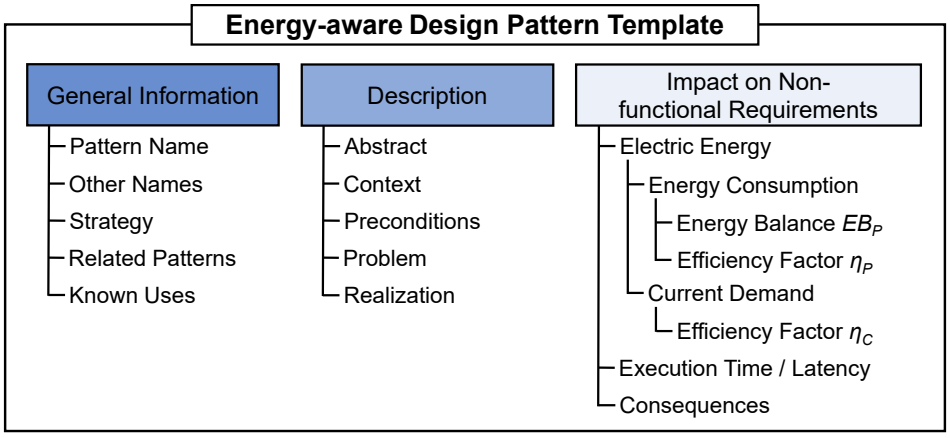


Figure 4.1: Energy-aware design pattern template structure.

Sections of the first two elements provide general information and an in-depth description of the energy-aware software design pattern similar to well-known design pattern templates as classical representations discussed in Section 2.4 (p. 32 ff.). However, none of the discussed design pattern templates considered the impact and side effects on NFPs related to power and energy consumption which are highly important for restricted or battery-powered systems. Since NFRs specify criteria based on NFPs and thus form the basis for evaluation, they should be considered in the high-level description of design patterns. Due to this, the new element *Impact on Non-functional Requirements* is introduced and provides sections describing the

impact and side effects on NFRs related to power, energy, and execution time. The remainder of this section explains the design pattern template with its elements and sections.

General Information

As part of the classical representation of design patterns, this element has been derived from [67, 131] and describes the meta-information of a design pattern by providing the following sections:

- *Pattern Name*: A name as a unique reference for the described design pattern.
- *Other Names*: Synonyms and well-known names for the design pattern, e.g., from other disciplines or domains.
- *Strategy*: A brief description of the basic strategic principles provided by the design pattern to address NFPs, e.g., power consumption. Developers may use this section for a broad filtering and selection process. A more in-depth description of the design pattern solution is provided in the *Description* section.
- *Related Patterns*: Names of other design patterns realizing the same or a closely related concept or solving related problems. Design patterns mentioned as related patterns may be combined with the described pattern. In addition, applying the described design pattern may result in the exclusion of other design patterns mentioned in this section. Therefore, this section provides the basis for an energy-aware design pattern language [10].
- *Known Uses*: Examples of existing solutions successfully using this design pattern, e.g., in other domains or disciplines such as electrical engineering.

Description

This element covers the basic definitions of the design pattern concept and the general conditions for effective use. The structure is derived from [21, 67, 131] and contains the following sections:

- *Abstract*: A short description of the pattern to provide a first overview.
- *Context*: Description of the situation where this pattern may be applied.
- *Preconditions*: Conditions including requirements and properties of the underlying hardware architecture, which must be fulfilled to apply the pattern successfully.
- *Problem*: Description of the problem addressed by this design pattern expressed as a question.
- *Realization*: Description of the implementation details. Depending on the pattern type, e.g., structural or behavioral design patterns, the level of detail in this section may vary. Typically, a textual description is provided. Due to diagram-based modeling languages such as UML (cf. Appendix A.3, p. 266 ff.), graphical representations can be added to the description. For structural energy-aware design patterns, structure, class, and object diagrams may extend the textual description. In contrast, state, timing, activity, or sequence diagrams may be added for behavioral patterns. SysML diagrams, such as the SysML block definition diagram, are also suitable for less detailed visualizations.

Impact on Non-functional Requirements

This novel element provides the following sections to describe the impact on NFRs related to electric energy and execution time:

- *Electric Energy*: This section is divided into two specializations with different sets of goals and metrics. Considering energy-aware systems (cf. Definitions 3.2–3.3, p. 85), energy-aware design patterns may address either the *energy consumption* or the *maximum current demand*. Additionally, each energy-aware design pattern description is extended with a standardized graphical representation to outline the energy-related and computational behavior of the defined design pattern. Figure 4.2 shows a variant of the graphical representation to illustrate the effects on energy consumption referred to as the power-timing diagram, first published by the author in [337]. In general, the graphical representation serves two purposes: On one hand, it visualizes the effects of design patterns uniformly. On the other hand, it helps to improve the understanding of each design pattern and thus eases the selection of suitable solutions. The basic structure of the power-timing diagram, as shown in Figure 4.2, consists of a two-part representation. The y-axis in the upper part of the diagram shows different power levels at which the system or a single hardware component may operate. The y-axis in the lower part of the diagram defines the computational power levels of the system or a single hardware component. Both parts of the diagram share the same x-axis representing the execution time t to illustrate the behavior over time. A black dashed line characterizes the behavior without applying the described design pattern. The red line shows the adapted behavior resulting from the energy-aware design pattern. Moreover, for the power level in the upper part of the power-time diagram (cf. Figure 4.2), the impact of the energy-aware design pattern is highlighted by green-filled areas if additional energy is required and by blue-filled areas to indicate potential energy savings. The graphical comparison of the

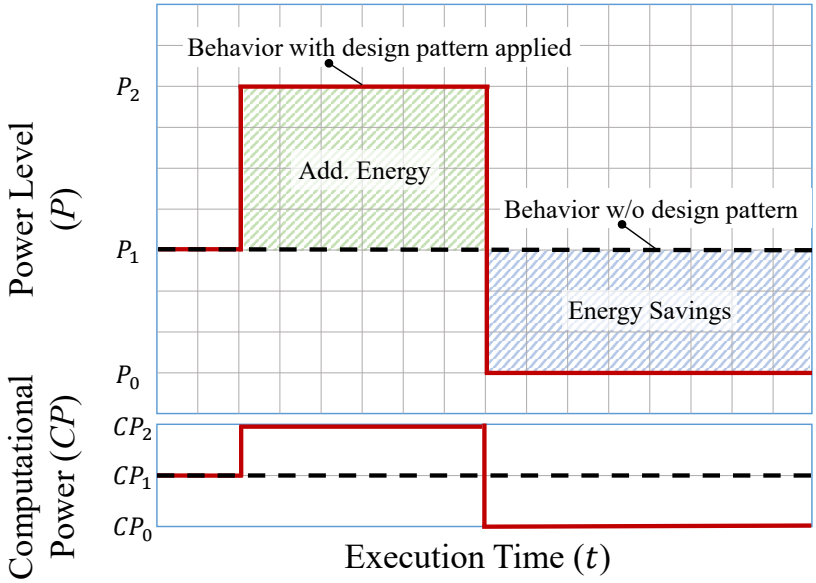


Figure 4.2: Power-timing diagram as a variation of the developed graphical representation, adapted from [337].

behavior with and without applied design patterns visualizes the properties, which are used by the metrics. This section is split into two subsections, as follows:

  - *Energy Consumption*: For this section, a set of metrics consisting of the *energy balance* EB_P and the *efficiency factor* η_P have been specified to describe the impact of design patterns on energy consumption uniformly. EB_P defines a balance equation to indicate possible energy savings, where a higher value of EB_P suggests more significant savings. η_P is derived from EB_P and describes an effort-saving ratio as a dimensionless factor ranging between 0 and 1. In this context, η_P enables a quantitative evaluation of the efficiency for possible energy savings resulting from applying an energy-aware design pattern. If $\eta_P = 1$, an energy-aware software design pattern saves energy without additional effort, whereas $\eta_P = 0$ means a design pattern is ineffective and does not save energy. Values within $]0, 1[$ describe a tradeoff between additional energy overhead and energy savings.
  - *Current Demand*: The effect of design patterns on the maximum current demand is expressed by the metric η_C. In general, η_C defines a cut-off factor expressed in percent, describing how much the current demand can be reduced. A value of 0 implies no improvement, while values for $\eta_C \geq 0.5$ can be considered a very effective improvement by the energy-aware design pattern. Note that the exemplary graphical representation in Figure 4.2 can also be applied to design patterns addressing the current demand. Such a representation is referred to as a current-timing diagram. It differs from a power-timing diagram only in the upper part, which describes the behavior related to current levels instead of power levels. However, since energy-aware design patterns referring to the current demand have other properties and metrics, the upper part might have a different structure.

- *Execution Time / Latency*: Describes the impact of the energy-aware design pattern on the execution time. This section also covers additional overhead and latencies due to the applied design pattern, e.g., by using worst, average, and best-case scenarios.

- *Consequences*: Describes tradeoffs when a design pattern is applied since optimizing one requirement may affect other requirements. In most of the pattern templates introduced in Section 2.4.1 (p. 32 ff.), this section deals with consequences related to development costs, portability, and modifiability. Design patterns addressing energy-related problems may have interrelations with other functional or non-functional properties [291]. Consequently, this section also contains a description of any disadvantages, drawbacks, and side effects introduced by the energy-aware design pattern, as well as changed or additional hardware requirements. This may lead to adaptations, which developers must address.

4.4 Energy-aware Design Pattern Catalog

This section provides the first catalog of energy-aware design patterns as a contribution to address RQ2. The lack of a catalog for design patterns targeting energy efficiency has also be identified by Bass et al. (2021) [38]. Energy-aware design patterns in this section are specified using the adapted design pattern template introduced in Section 4.3. The pattern catalog contains a total of six energy-aware design patterns with five design patterns based

on existing, well-known, and proven principles and one novel design pattern originating from own research published in [337, 392]. If not already done, the energy-aware design patterns presented were identified through the design pattern identification process and described using the new design pattern template proposed in Section 4.3 (p. 91 ff.). Their general validity is described by the corresponding definitions and equations. The examples are given for the sake of understandability and do not restrict design patterns in their use. Numerical values for the examples are assumed or taken from further literature. However, this section focuses on the fundamental and uniform consideration of best practices from an energy perspective as well as evaluation and efficiency measures. Energy-aware design patterns may be organized according to their main category, as shown in Table 4.1.

Design Pattern	**Classification**	**Category**		
		Utilization	*Res Mgmt.*	*Concurrency*
Energy-aware Sampling	behavioral	✓		
Event-based Computing	behavioral	✓		
Power Monitor	behavioral		✓	
Direct Memory Access Delegation	behavioral	✓		
Mirroring	behavioral			✓
Race-To-Sleep	behavioral	✓		✓

Table 4.1: Classification and Categorization of six energy-aware design patterns.

According to Table 4.1, most of the energy-aware design patterns address the control and utilization of individual hardware components during runtime (*Utilization*). Two of the design patterns deal with *concurrency* aspects to improve the energy efficiency of the embedded system, while only the *Power Monitor* design pattern focuses on resource management (*Res. Mgmt.*) of peripheral devices such as sensors and actuators. Although the introduced design pattern template is able to specify energy-related aspects of a software application from different perspectives, energy-aware design patterns presented in this section focus exclusively on energy consumption. However, future work may expand the design pattern catalog with design pattern descriptions focusing on electric current demand.

Each of the following energy-aware design pattern descriptions contains a uniform power-timing diagram as a graphical representation of the behavior related to power consumption, computational power, and execution time, as introduced in the previous Section 4.3 (p. 91 ff.). Based on the characteristics illustrated by the power-timing diagram, the impact of design patterns is described with the proposed metrics EB_P and η_P.

4.4.1 Energy-aware Sampling (EAS)

This section describes the *Energy-aware Sampling (EAS)* design pattern.

General Information

Other Names: *Adaptive Sampling* as used in literature, e.g., in [356].

Strategy: A (unnecessary) high sampling rate increases the power consumption of a hardware component due to a longer active period. EAS influences the *time* a peripheral device is operating in an active state. By lowering the sampling rate, a peripheral device can be inactive

for a longer period of time, and the CPU can enter a lower power state. This pattern may also be applied on CPUs to switch periodically between active and low-power states.

Related Patterns: *Cost-Aware Sampling* and *Quality of Service Based Sampling* as two specializations of the generic EAS pattern are described in [253]. The first pattern adjusts the sampling rate according to the sampling cost based on power consumption, memory size, and communication bandwidth, while the second pattern affects the sampling rate based on the transmission network performance.

Known Uses: A dynamically adapting sampling frequency is used in [356] to save approximately 31 % of the system's battery energy during a three months continuous water quality monitoring period.

Description

Abstract: The sampling rate has a strong impact on the energy consumption of the system [388]. The main strategy of EAS is to adjust the sampling rate of peripheral devices in an energy-aware. This is achieved by selecting a sampling rate that fits the relevant frequencies to extract all the necessary information.

Context: EAS is highly suitable for periodic systems with constant sampling rates. It may be used for peripheral devices in situations where signal characteristics are known and algorithms for data processing can handle varying sampling rates. For CPUs, EAS can be applied when the execution of the software application contains a high percentage of idling time.

Preconditions: Peripheral devices addressed by this pattern must have the capability to adjust the sampling rate. In addition, the characteristics of the signal must be well known. When used in combination with a CPU, power state changes at runtime must be supported.

Problem: How can the energy consumption of a system be optimized by adjusting the sampling rate of a peripheral device or by reducing the period of the active state of a CPU?

Realization: A static adjustment of the sampling rate for peripheral devices can be achieved during the startup process of the software application. If the adjustment is supposed to vary during runtime, further software components, e.g., algorithms or data transmissions, need to be considered. If EAS is used for peripheral devices such as sensors, the lowest sample frequency f_{sample} should be $f_{sample} > 2 \cdot f_{max}$, with f_{max} as the maximum frequency of the signal, to extract all the necessary information, also known as the Nyquist-Shannon sampling theorem [211]. To adjust the active time of CPUs, parts of the software application containing idling times need to be identified. For those parts, the CPU can be set to a lower power state. In general, EAS has a minimal impact on the software application and can be implemented rapidly. In the case of a static adjustment, this ideally takes place at the initializing phase of the software application. If the sampling interval varies dynamically at runtime, further software components may need to be adapted.

Impact on Non-functional Requirements

Figure 4.3 shows the power-timing diagram for EAS applied to lower the energy consumption of a CPU without considering peripheral devices and sensors.

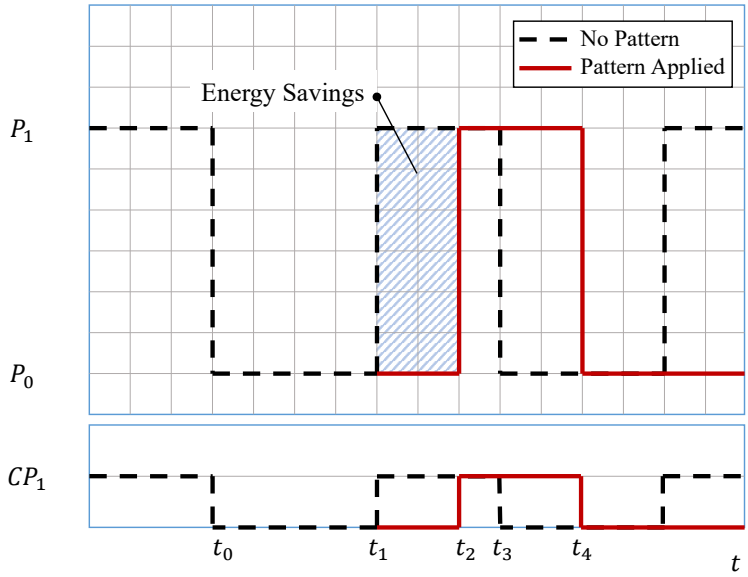


Figure 4.3: Power-timing diagram for the EAS design pattern (published in [337]).

Energy Consumption: The power states P_i are defined in the upper part of Figure 4.3, with the sleep mode as P_0 and the normal mode as P_1. The lower part represents the computational power CP, where CP_1 describes the computational power in normal mode. The power (duty) cycle D is defined as:

$$D = \frac{c}{T} \tag{4.1}$$

with the duration c as the duration, the CPU operates in normal (active) mode and T as the period consisting of a sleep phase and an active phase defined as:

$$c = t_3 - t_1 \tag{4.2}$$
$$T = t_3 - t_0 \tag{4.3}$$

When EAS is applied, a new period $T' > T$ is defined, leading to a new relaxed power cycle D', defined as:

$$D' = \frac{c}{T'} \tag{4.4}$$
$$T' = t_4 - t_0 \tag{4.5}$$

Note that the duration c in Equation (4.1) is unchanged while the new period T' is increased and thereby extends the time the CPU operates at P_0. EB_P, as defined in Equation (4.6), can be calculated using Equations (4.1) to (4.5) and $\Delta P_{10} = P_1 - P_0$. When taking other

peripheral devices into account, Equation (4.6) needs to be extended.

$$\begin{aligned} EB_P &= E_{normal} - E_{relaxed} \\ &= (D \cdot \Delta P_{10}) - (D' \cdot \Delta P_{10}) \\ &= \Delta P_{10}\ (D - D') \end{aligned} \tag{4.6}$$

The relationship between relaxing the duty cycle and energy savings is linear. Since additional power or computational power effort is not required, the efficiency factor can be defined as $\eta_P = 1$.

Execution Time / Latency: When adapting the duty cycle, the execution time of a software application may be affected. For example, periodic latencies where the system waits until measured values of a sensor are obtained may be reduced since the sensor is used less often during the same time period.

Consequences: When using EAS to adjust the reading of a sensor, the number of total data points decreases. EAS may also have an impact on the accuracy of the sampled signal due to the reduced sampling rate. Reducing the sampling rate of peripheral hardness or the active phase of a CPU may increase response times.

4.4.2 Event-based Computing (EBC)

This section describes the *Event-based Computing (EBC)* design pattern.

General Information

Other Names: None known.

Strategy: CPUs can achieve low energy consumption by minimizing the time spent in active mode and maximizing the time operating in low-power mode. EBC reduces the *active time* of CPUs by using interrupts instead of polling loops. The CPU, which would otherwise only be active to execute polling loops with subsequent data processing, can be set into a low-power mode for the time between interrupts.

Related Patterns: A similar approach referred to as *Event-Triggered Sampling* is presented in [253], where events are used to reduce the communication between devices by avoiding the transmission of redundant data and, thus, reducing the number of transmitted messages.

Known Uses: In event-based development, internal and external interrupts can be used to trigger specific functions of the software application causing spontaneous behavioral changes. Peripheral devices like ADCs and external devices, such as the NXP CLRC663 plus NFC frontend [268], use interrupts to signal the host processor when thresholds are reached or changes are detected.

Description

Abstract: The EBC design pattern optimizes the power consumption of a CPU by replacing polling loops in the software application with interrupt implementations. Interrupts may be

used by internal and external peripheral devices to indicate state changes directly. They serve as triggers for events and cause a spontaneous change in the state of a software application. By this, peripheral devices may operate independently of the CPU and, for example, use interrupts to signal the presence of new data.

Context: This pattern can be applied to software applications that need to respond to spontaneous events caused by peripheral devices. EBC is also suitable for time-critical systems.

Preconditions: This pattern requires peripheral devices with interrupt support and built-in trigger functionalities, e.g., ADCs combined with comparators or external peripheral devices with built-in interrupt support.

Problem: How can a system process discrete events but remain in a low-power state most of the time otherwise?

Realization: Polling forces the application to query peripheral devices constantly. This behavior produces wait cycles and keeps the CPU active, leading to a significantly increased power consumption. Avoidable wait cycles can be replaced by interrupts and *Interrupt Service Routines (ISRs)*. A software application can direct the CPU to enter a low-power mode and configure peripherals to wake up the CPU if necessary.

Impact on Non-functional Requirements

The basic power characteristics of the EBC design pattern are outlined by the power-timing diagram shown in Figure 4.4.

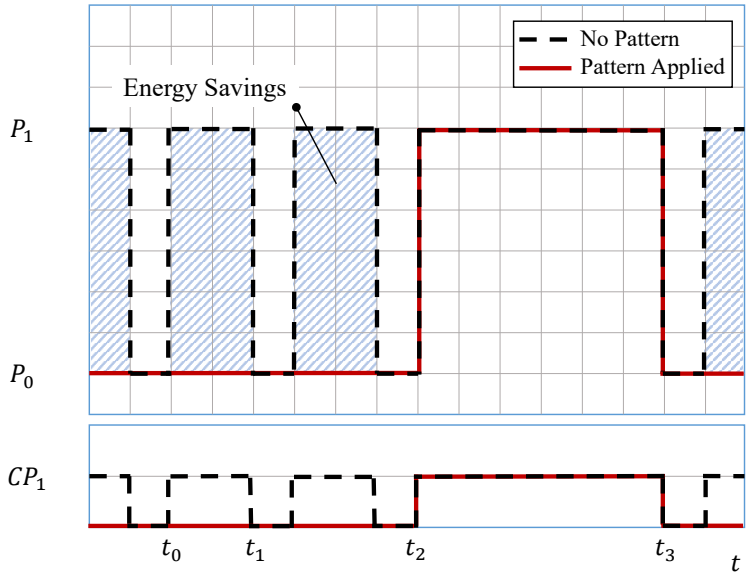


Figure 4.4: Power-timing diagram for the EBC design pattern (published in [338]).

Energy Consumption: The upper part of Figure 4.4 contains the power levels P_0 for the low power mode and P_1 for the active mode of a CPU. When no pattern is applied, the

application in the example represented in Figure 4.4 constantly queries a peripheral device at a fixed interval. For a better understanding of the behavior, the terms *miss* and *hit* are introduced. A *miss* defines a polling operation where the result does not lead to a behavioral change of the software application, for example, a follow-up data processing activity. In case of a *hit*, e.g., a threshold is exceeded. As a result, the software application obtains the data from the peripheral device and starts (computationally intensive) data processing operations. W.r.t. the power-timing diagram shown in Figure 4.4, *miss* and *hit* can be specified by:

$$\Delta t_{miss} = t_1 - t_0 \tag{4.7}$$

$$\Delta t_{hit} = \Delta t_{interrupt} = t_3 - t_2 \tag{4.8}$$

When this pattern is applied, the CPU will be notified and can query peripheral devices if necessary and directly start (computationally intensive) data processing operations. To reduce the complexity of the power-timing diagram shown in Figure 4.4, the duration of a *hit* (no pattern) and interrupt solution (with pattern) are equal in length, cf. Equation (4.8). To describe the behavior of the system even further, the following definitions are required:

$$\Delta P_{10} = P_1 - P_0 \tag{4.9}$$

$$E_{polling} = n \cdot (\Delta t_{miss} \cdot \Delta P_{10}) + \Delta t_{hit} \cdot \Delta P_{10} \tag{4.10}$$

$$E_{interrupt} = \Delta t_{hit} \cdot \Delta P_{10} \tag{4.11}$$

$E_{polling}$ in Equation (4.10) defines the energy if polling is used with the parameter n as the number of polling requests which do not lead to follow-up data processing operations (*miss*). In Equation (4.11), $E_{interrupt}$ defines the expected energy consumption when the design pattern is applied. Energy savings result from avoiding such polling requests. The resulting energy balance EB_P can be calculated as follows:

$$\begin{aligned} EB_P &= E_{polling} - E_{interrupt} \\ &= n \cdot (\Delta t_{miss} \cdot \Delta P_{10}) \end{aligned} \tag{4.12}$$

The *efficiency factor* is $\eta_P = 1$ since the appliance of this pattern does not produce additional power-related overhead if $\Delta t_{interrupt} = \Delta t_{hit}$, as defined in Equation (4.8). If $\Delta t_{interrupt} > \Delta t_{hit}$, η_P can be calculated as follows:

$$\eta_P = 1 - \left(\frac{E_{interrupt}}{E_{polling}} \right) \tag{4.13}$$

Execution Time / Latency: Adapting the strategy of the software application from a polling-based to an event-based approach has no negative effect on time behavior. Event-driven software applications are suitable for real-time requirements. In the provided example, an additional overhead of the CPU based on context switching when handling interrupts is not considered. However, context switching is use case specific and depends strongly on the CPU architecture, CPU model, and selected low-power mode.

Consequences: Interrupts are generally causing changes in the workflow and structure of the application, which has to be considered. GPIO-based interrupts may require additional wires or lines, which may lead to hardware design changes.

4.4.3 PowerMonitor

This section describes the *PowerMonitor* design pattern, which is based on previous work [392].

General Information

Other Names: None known.

Strategy: The *PowerMonitor* design pattern optimizes the *time* a peripheral device stays in an active state and lowers the *electrical capacitance* of the system. The design pattern introduces an additional abstraction layer for hardware accesses. All peripheral devices and interfaces can be dynamically controlled and, for instance, automatically disabled if they are no longer in use or requested by any object of the software application. By this, uncontrolled access to shared resources, e.g., through competing tasks, is also addressed.

Related Patterns: Power-gating [188], where the principle of the *PowerMonitor* design pattern is used at the block level in the integrated circuit design of the hardware layer.

Known Uses: A proof-of-concept implementation of the *PowerMonitor* design pattern is presented in [392].

Description

Abstract: The *PowerMonitor* design pattern considers the power consumption properties of internal peripheral devices of SoCs and external peripheral devices. This also includes communication interfaces such as I^2C and SPI with various connected devices. As a centralized approach, the *PowerMonitor* design pattern has deep knowledge of devices and communication interfaces at runtime and is able to dynamically change their power modes when they are temporarily not needed. By this, energy-efficient systems may be defined without loss of functionality.

Context: The *PowerMonitor* design pattern may be applied to software applications, which must periodically access peripheral interfaces and devices. Moreover, the design pattern may also be suitable if a centralized and fine-grained hardware access control has to be achieved.

Preconditions: Depending on the intended usage, the software application requires control over all considered communication interfaces, such as I^2C or SPI, and the capability to disable and enable external devices, e.g., sensors and actuators, as well as clocks of functional units.

Problem: How can a software application with a fine-grained power-saving strategy be implemented, which only enables peripheral devices on request? Additionally, how can conflicts between sleep modes (e.g., preventing software from being executed) and use cases with continuous tasks be addressed?

Realization: The reference implementation of the PowerMonitor design pattern [392] follows a template meta-programming approach using *C++17* to provide an abstract and type-safe interface. The overall structure and involved components of the *PowerMonitor* design pattern are outlined in Figure 4.5. The access of communication interfaces, e.g., I^2C or SPI, and

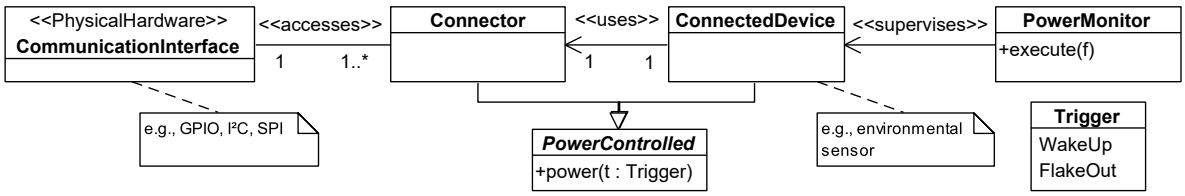


Figure 4.5: Structure and components of the *PowerMonitor* design pattern reference implementation (adapted from [337, 338, 392]; UML 2.5 class diagram notation).

peripheral devices are managed by a single *PowerMonitor* class instance. Other parts of the software application may use the provided functions of the *PowerMonitor* class and do not have to implement hardware accesses themselves. The components shown in Figure 4.5 are:

- `PowerMonitor`: This class defines the flow control f_c within the software application and has access to connected peripheral devices. When active, required power-controlled communication interfaces and peripheral devices are enabled. When the monitor function is exited, peripheral devices are placed into sleep mode automatically.
- `PowerControlled`: This entity supports some kind of power management. This class has to implement a function that accepts power triggers as wake-up and flake-out signals.
- `ConnectedDevice`: This class provides an abstract interface to the user part of the software application, abstracts the used hardware component, and specializes `PowerControlled`.
- `Connector`: This class is a low-level abstraction to provide basic functions of `CommunicationInterfaces`.
- `CommunicationInterface`: This class is a communication interface provided by the MCU, e.g., GPIO or bus systems such as I^2C or SPI, or CAN.

The `execute` method of the *PowerMonitor* is called from the application layer in order to get access to a peripheral device. This triggers the `power` method of a `ConnectedDevice` instance using a wake-up signal as a parameter, which is also propagated to the linked communication interface. When all operations have been completed, the `PowerMonitor` instance sends another trigger, which initiates the flake-out phase of the concerned objects. By this, a `PowerMonitor` instance defines a specific section within the software application for controlled access to communication interfaces and peripheral devices. Outside this section the device moves dynamically to a low-power or sleep mode reducing the system power.

Impact on Non-functional Requirements

The power-timing diagram pictured in Figure 4.6 sketches the characteristics of the *PowerMonitor* design pattern.

Energy Consumption: For the scenario introduced by the power-timing diagram in Figure 4.6, it is assumed that the considered communication interface and the peripheral device are disabled before t_0. This state is denoted as power consumption level P_1. At first, the communication interface to access the peripheral device, e.g., I^2C, is enabled at t_0. Afterward, at

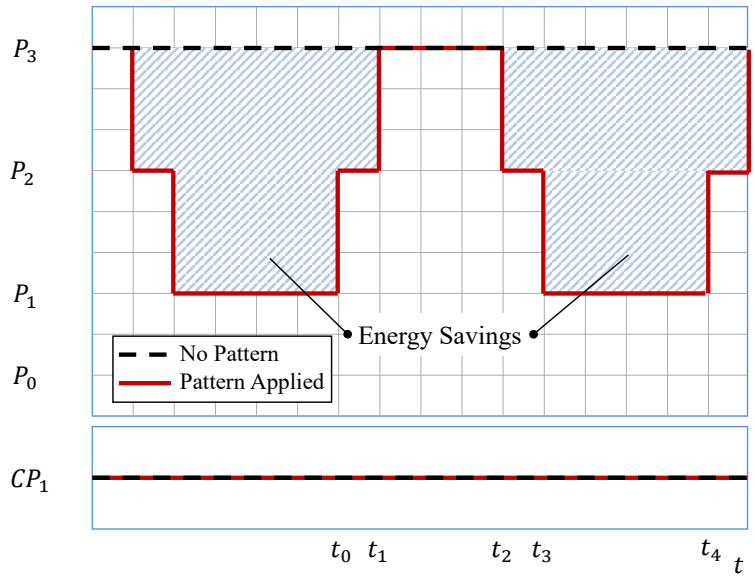


Figure 4.6: Power-timing diagram for the *PowerMonitor* design pattern (published in [337]).

t_1, the peripheral device, e.g., a connected sensor, gets enabled. To describe the behavior of the system and the software application on which the *PowerMonitor* design pattern has been applied, the following definitions are required referring to Figure 4.6:

$$T = t_4 - t_0 \tag{4.14}$$

$$\Delta t_{10} = t_1 - t_0, \qquad \Delta t_{21} = t_2 - t_1, \qquad \Delta t_{32} = t_3 - t_2 \tag{4.15}$$

and

$$\Delta P_{21} = P_2 - P_1, \quad \Delta P_{32} = P_3 - P_2, \quad \Delta P_{31} = P_3 - P_1 \tag{4.16}$$

For the time frame Δt_{21}, the application can use the device without any loss of functionality, and the power consumption level is P_3. Afterward, the `PowerMonitor` instance dynamically disables the external device and the communication interface while the power consumption drops back to the previous level P_1.

Possible energy savings can be calculated using Equations (4.15) to (4.16) which strongly depend on the power consumption of the communication interface defined as ΔP_{21} and the external device defined as ΔP_{32}. The energy consumption for the software application, without (E_{normal}) and with ($E_{monitor}$) the design pattern applied, may be calculated as:

$$E_{normal} = T \cdot \Delta P_{31} \tag{4.17}$$

$$E_{monitor} = \underbrace{(\Delta t_{10} + \Delta t_{32}) \cdot \Delta P_{21}}_{\approx 0} + \Delta t_{21} \cdot \Delta P_{31} \tag{4.18}$$

The first part of $E_{monitor}$ can be considered very small and close to zero since enabling functional units usually takes only a few clock cycles. Equation (4.19) provides the calculation of the energy balance EB_P:

$$\begin{aligned} EB_P &= E_{normal} - E_{monitor} \\ &\approx \Delta P_{31} \cdot (T - \Delta t_{21}) \end{aligned} \tag{4.19}$$

The computational power shown in the lower part of Figure 4.6 is unaffected by this pattern and remains constant at level CP_1. The efficiency factor is $\eta_P = 1$ because the basic concept does not require additional energy.

Execution Time / Latency: Latencies may be introduced by the additional overhead of the software application to manage and control devices. Additional latencies may occur during switching the power states of affected hardware devices since the response time of those devices may be increased compared to devices continually running. However, these type of latencies cannot be generalized since they depend on the specific implementation of the software application and hardware layer characteristics. Therefore, they are not part of the power-timing diagram pictured in Figure 4.6.

Consequences: This design pattern requires a deep knowledge of each communication interface and peripheral device, as well as their energy consumption characteristics, which increases the complexity of the software application design. Additionally, for more complex peripheral devices, e.g., network adapters, the recreation of its contexts requires additional application logic if, for example, the connection is lost due to a timeout. Energy consumption may be reduced without any loss of functionality depending on the characteristics of communication interfaces and peripheral devices. Enabling and disabling hardware components periodically may add further latencies, as mentioned in the *Execution Time / Latency* field. In some cases, the startup phase of such devices may be more energy expensive, for instance, due to inrush current, compared to steady-state operating devices. Additionally, changing the states in short intervals may have a negative impact caused by re-establishing phases of a radio connection or the preheating phases of gas sensors. Using interrupts for asynchronous events may also be limited if clocks have been disabled.

4.4.4 Direct Memory Access Delegation (DMAD)

This section describes the *Direct Memory Access Delegation (DMAD)* design pattern.

General Information

Other Names: None known.

Strategy: Transferring a large amount of data may be time-consuming, depending on the clock settings of the communication bus used, the source's reading speed, and the destination's writing speed. During data transfer, the core of a CPU operates in active mode. The strategy of this pattern is to use the DMA method to handle data and memory transfers without using a CPU core, which can instead be put into a low-power mode to reduce power consumption. This may also result in a lower overall *electrical capacitance* if devices can be turned off completely. Because a DMA-based transfer is typically faster, the active *time* of a system is also reduced.

Related Patterns: None known.

Known Uses: *High Speed Serial Port* [115] describes a hardware interface design pattern where DMA is used to transfer data between a serial device and memory without CPU intervention.

Description

Abstract: The DMAD pattern optimizes the energy consumption of a system by using DMA for data transfers between peripheral devices and memory units. The CPU core, which would otherwise be responsible for the communication, can be set to a lower power mode during data transfer.

Context: This design pattern is highly suitable for use cases where larger data transfers or continuous data streams need to be processed automatically. The CPU core can be set to a power-saving state if no other application load has to be calculated for the data transfer period, for example, since a DMA controller operates in parallel with the CPU, allowing autonomous communication between peripheral devices without requiring CPU clock cycles. This pattern should also be considered if a high-speed data transmission up to 100 Mbps is needed.

Preconditions: This design pattern requires a system with DMA support.

Problem: How can an energy-efficient data transfer between peripheral devices or memory units be achieved without using the CPU?

Realization: At its core, DMAD is independent of the software application but depends on the hardware platform and the wiring of the DMA. DMA controllers and interrupts are typically configured during the initialization phase of the software application. As a result, the software application requires adaptions to include such platform-specific configuration commands. The CPU only has to respond to those interrupts and ISRs, which, for example, are triggered when a data transfer task is finished. Common use cases are audio and video data streams and applications where continuous ADC values are required, which can be directly transferred into the memory or to other peripheral devices.

Impact on Non-functional Requirements

Figure 4.7 shows the power-timing diagram for the DMAD design pattern.

Energy Consumption: The *power-timing diagram* shows a simplified use case for a basic understanding of the impact on power consumption and time behavior. The upper part of Figure 4.7 shows the temporal behavior and the power consumption levels of the pattern. P_0 is defined as the state where the CPU core and the DMA controller are operating in a low-power mode. Assuming that the power consumption of the DMA controller is lower than the power consumption of the CPU, P_1 defines the power state where the DMA controller is active, while the state P_2 defines the power state of the system where the CPU is operating. For simplification, the configuration overhead and possible power consumption when the DMA operates in an idle state are ignored. To calculate the energy balance EB_P, the following equations are defined with reference to Figure 4.7:

$$\Delta t_{10} = t_1 - t_0, \quad \Delta t_{20} = t_2 - t_0, \quad \Delta t_{21} = t_2 - t_1 \tag{4.20}$$

and

$$\Delta P_{10} = P_1 - P_0, \quad \Delta P_{20} = P_2 - P_0 \tag{4.21}$$

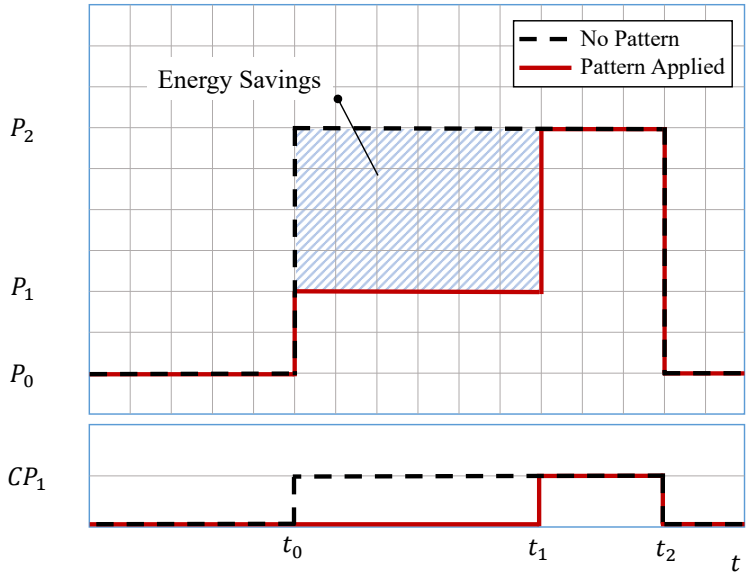


Figure 4.7: Power-timing diagram of the DMAD design pattern (published in [338]).

For this pattern, the energy balance EB_P can be calculated by subtracting the energy consumption when the DMAD pattern has been applied (E_{dma}) from the solution without the design pattern (E_{normal}) using Equations (4.20) to (4.21) as follows:

$$\begin{aligned} EB_P &= E_{normal} - E_{dma} \\ &= (\Delta t_{20} \cdot \Delta P_{20}) - (\Delta t_{10} \cdot \Delta P_{10} + \Delta t_{21} \cdot \Delta P_{20}) \\ &= \Delta t_{10} \cdot (\Delta P_{20} - \Delta P_{10}) \end{aligned} \tag{4.22}$$

Equation (4.22) indicates that the DMA must be faster or consume significantly less power than the CPU to apply the design pattern effectively. Since the design pattern does not require additional power or computational power effort, the *efficiency factor* can be defined as $\eta_P = 1$.

Execution Time / Latency: The DMAD design pattern may accelerate the data transfer due to the direct DMA communication between peripheral devices without the CPU being involved. Furthermore, the CPU core can use the number of cycles saved for other tasks or be set to a low-power mode.

Consequences: An additional overhead is caused by the configuration of the DMA controller. For example, if only a few data values have to be read from the ADC, additional CPU cycles to set up the DMA controller neutralize the effect of the DMAD design pattern. In addition, the structure of the software application has to be reviewed to support DMA and interrupts.

4.4.5 Mirroring

This section describes the *Mirroring* design pattern.

General Information

Other Names: None known.

Strategy: This design pattern describes strategies to lower the *execution time* of a system by shifting the application (partly) between CPU cores at runtime. The *Mirroring* design pattern can be used for two different strategies. In the first strategy, the execution time of computation-intensive parts can be reduced by migrating from an energy-efficient to a high-performance core. When a significant amount of idle time exists, tasks can be moved from a high-performance to an energy-efficient core as a second strategy.

Related Patterns: If multiple CPU cores are used simultaneously for a short period of time to finish the workload earlier, it corresponds to the *Race-To-Sleep* design pattern as described in Section 4.4.6 (p. 111 ff.).

Known Uses: ARM's big.LITTLE describes a hardware-based technology for heterogeneous multiprocessor architectures, which can be seen as a hardware implementation of the *Mirroring* pattern. The architecture allows tasks to be assigned to a high-performance or energy-efficient core depending on the expected computational intensity [420]. At runtime, the running state of tasks can be transferred between CPU cores.

Description

Abstract: The *Mirroring* design pattern is able to migrate an application or parts of the workload, e.g., defined as tasks, between CPU cores with different power levels and power characteristics at runtime.

Context: Use the *Mirroring* design pattern if the underlying system consists of a multi-core architecture and the software application can influence the execution environment of tasks dynamically at runtime.

Preconditions: This design pattern requires a specific CPU or system architecture. Each CPU or MCUcore must be able to communicate with other cores, e.g., via signaling and inter-core communication, and must be able to change the operating mode or adjust the operating frequency at runtime.

Problem: How can a software application or parts of the software application, e.g., tasks, be switched dynamically between individual cores of a multi-core CPU during runtime to increase energy efficiency?

Realization: The concept of the *Mirroring* design pattern can be applied to different CPU and system architectures. Note that with technologies such as ARM's big.LITTLE, tasks may be switched between cores without the need to extend parts of the software design. In the absence of low-level hardware support, a management layer that controls and coordinates the

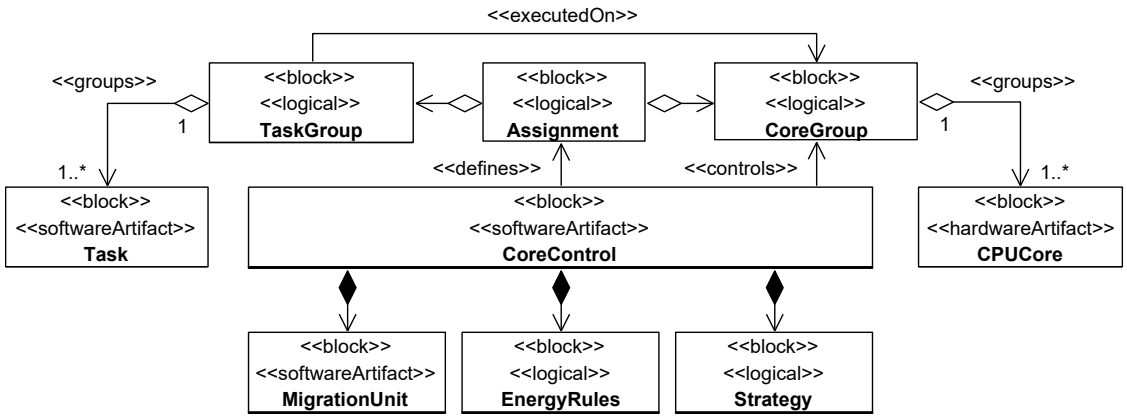


Figure 4.8: Exemplary software design using the *Mirroring* design pattern (adapted from [337]; (SysML 1.6 block definition diagram notation).

tasks must be developed. An example of a software-based implementation for multi-core CPUs is shown in Figure 4.8. The elements of the *Mirroring* design pattern example in Figure 4.8 have the following characteristics:

- `TaskGroup`: The class defines a group of `tasks` that can be executed on a `CoreGroup`.
- `CoreGroup`: This is a group of CPU cores.
- `CoreControl`: This block manages `CoreGroups` and defines `Assignments` between a `TaskGroup` and a `CoreGroup`. It implements functions for measuring the load of the CPU cores and consists of a `Strategy`, a `MigrationUnit`, and a set of `EnergyRules`.
- `Strategy`: This block defines the execution and association rules of `cores` and `tasks`, e.g., the order in which tasks are migrated and cores powered. Depending on the CPU architecture and abilities of the operating system used, a `TaskGroup` may be transferred between `CoreGroups` without additional implementation effort. If the running state of a task can not be transferred, another strategy uses, for instance, a second task denoted as the mirroring task implementing the same functionality as the original task. It is assigned to another CPU core, typically on hold, and only executed if a migration by the `CoreControl` is initiated.
- `MigrationUnit`: This class provides functions to instruct the cores and perform migrations. It also includes task management and context-switching strategies along with power management functions.
- `EnergyRules`: This class contains rules based on CPU registers, performance counters, and NFPs of abstract core representations in the `CoreGroup`, which the `CoreControl` may use for an energy-aware processing.

Impact on Non-functional Requirements

Figure 4.9 outlines the power-timing diagram for the *Mirroring* design pattern.

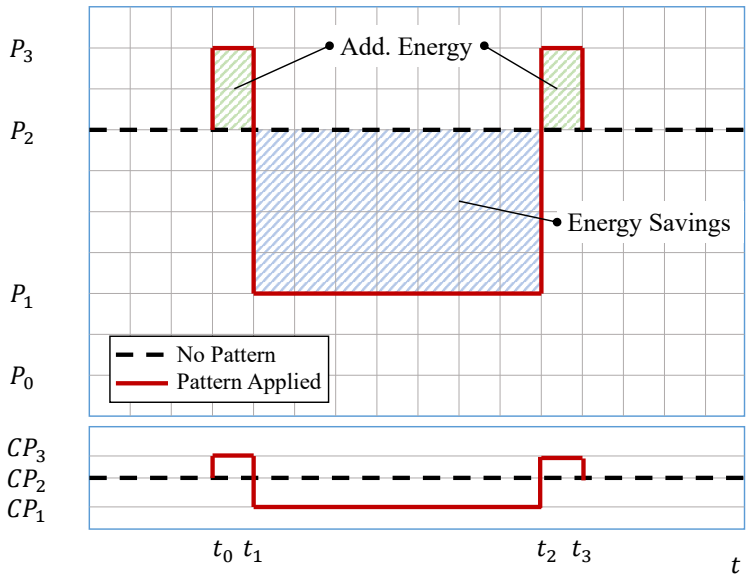


Figure 4.9: Power-timing diagram of the *Mirroring* design pattern for a dual-core CPU (published in [337]).

Energy Consumption: The purpose of this pattern is to optimize power consumption and energy efficiency by dynamically controlling different cores of a CPU or multiple CPUs of a system, as mentioned in the field *Strategy*. The impact on power consumption depends on how the cores are controlled. Figure 4.9 shows the *power-timing diagram* for a dual-core CPU consisting of an energy-efficient core and a high-performance core. The use case shown in Figure 4.9 describes the migration of a task running on a high-performance core to an energy-efficient core. The upper part of Figure 4.9 describes the power consumption P of the processor. The power state P_0 represents the state of the system when both cores operate in low-power mode, denoted as sleep mode. P_1 is reached when only the energy-efficient core is active, P_2 when only the high-performance core is active, and P_3 when both cores are active at the same time.

The computational power CP is defined as CP_1 for the energy-efficient core, CP_2 for the high-performance core, and CP_3 as the computational power when both cores are active simultaneously. The mirroring of a task in Figure 4.9 starts at t_0. The application moves relevant tasks from the high-performance to the energy-efficient core. During this time frame, both cores are active and causing a power consumption overhead. At t_1, the high-performance core is set into the defined low-power mode. For the time between t_1 and t_2, the system utilizes the low-power core. At t_2, tasks are shifted back to the high-performance core, resulting in an additional power overhead for the time between t_2 and t_3. To calculate the energy balance

EB_P and the efficiency factor η_P, the following definitions are required referring to Figure 4.9.

$$\Delta t_{10} = t_1 - t_0, \quad \Delta t_{21} = t_2 - t_1, \quad \Delta t_{32} = t_3 - t_2 \tag{4.23}$$

$$\frac{\Delta t_{10} + \Delta t_{32}}{\Delta t_{21}} = q_t < 1 \tag{4.24}$$

In Equation (4.24), q_t defines the quotient of the switching duration between the two cores and the time the energy-efficient core is active. To be effective, the total execution time of the energy-efficient core must be greater than the duration for switching between cores, and thus the $q_t < 1$. In Equation (4.26), q_P is defined as the quotient between the power consumption overhead required for the context switch where both cores are active and the power consumption if only the high-performance core is active. It is assumed that the power consumption of the energy-efficient core is lower than the power consumption of the high-performance core $q_P < 1$.

$$\Delta P_{21} = P_2 - P_1, \quad \Delta P_{32} = P_3 - P_2 \tag{4.25}$$

$$\frac{\Delta P_{32}}{\Delta P_{21}} = q_P < 1 \tag{4.26}$$

Equations (4.23) to (4.26) may be used to calculate the energy saved (E_{save}) and additional energy (E_{add}) when applying the design pattern. Assuming that $\Delta t_{32} = \Delta t_{10}$, the energy balance EB_P can be calculated as follows:

$$\begin{aligned} EB_P &= E_{save} - E_{add} \\ &= \Delta P_{21} \cdot \Delta t_{21} - ((\Delta P_{32} \cdot \Delta t_{10}) + (\Delta P_{32} \cdot \Delta t_{32})) \\ &= \Delta P_{21} \cdot \Delta t_{21} - 2 \cdot (\Delta P_{32} \cdot \Delta t_{10}) \\ &= \Delta P_{21} \cdot \Delta t_{21}(1 - q_P q_t) \end{aligned} \tag{4.27}$$

The efficiency factor for the *Mirroring* design pattern is specified as $\eta_P = (1 - q_P q_t)$. If, for example, $q_p = 0.125$ and $q_t = 0.1$, the efficiency factor can be calculated as $\eta_P = 0.988$. In general, the efficiency of the *Mirroring* design pattern highly depends on the application's workflow and the characteristics of the CPU. Energy is consumed for each state change, e.g., entering a low-power mode, due to the process of loading and unloading transistors, which has to be considered in the application design [396].

Execution Time / Latency: This pattern impacts the execution time in two ways: The first impact results from executing additional software application parts to manage tasks and CPU cores. The migration of a task itself causes the second impact. If a task, for example, is moved between two differently clocked cores, the execution time may be shortened or extended.

Consequences: This pattern can be modified to assign m different tasks to n cores with $m \geq n$ with $m, n \in \mathbb{N}$. Development costs are low if CPUs or cores share the same architecture, compiler, and programming language. If source code for a task has to be ported from, e.g., C++ to ASM, the development costs will increase due to the introduced required knowledge, increased complexity, and up to n variants for the same task to be implemented. For the reference implementation, the `CoreControl` class represents the managing instance, which must be executed on a core that is either permanently active or switched off last. To get the best energy balance, the most energy-efficient core is typically selected, which may introduce other disadvantages due to the reduced computational power.

4.4.6 Race-To-Sleep

This section describes the *Race-To-Sleep* energy-aware design pattern.

General Information

Other Names: Race-To-Idle, Race-To-Halt, Race-To-Zero, Race-To-Black.

Strategy: The *Race-To-Sleep* design pattern may be applied in two basic variants to reduce the *execution time* and, thus, lower the overall energy consumption of a system. In single-core environments, the CPU uses the highest possible operating frequency to compute the application workload as fast as possible. Afterward, the CPU switches to a low-power state to save energy. The second variation addresses multi-core environments, where the software application may be split and executed on different CPU cores to speed up processing, reducing the energy demand. Note that both variants may be combined.

Related Patterns: When applied in single-core environments, the concept of the *Race-To-Sleep* design pattern is similar to DFS [300], where the frequency of a CPU can be adjusted during runtime depending on the actual requirement.

Known Uses: The basic concept of this pattern is used for speed scaling in [9]. A multi-core scenario where the software applications may (partially) benefit from parallel processing to reduce the execution time is described in [88, 328].

Description

Abstract: This design pattern can affect the dynamic and static power consumption of the system, as described in Section 2.1.1 (p. 17 ff.). Computation-intensive software applications, in particular, can profit from the *Race-To-Sleep* design pattern. For single-core use, the highest possible operating frequency may be configured. In multi-core environments, the software application may executed on different CPU cores.

Context: *Race-To-Sleep* may be used when software applications are computationally intensive or contain computationally intensive sections.

Preconditions: In single-core environments, the CPU must be able to adjust the frequency at runtime. To achieve parallel processing in multi-core configurations, developers must ensure that the software application can be (partly) parallelized and does not induct bottlenecks due to a high proportion of sequential and non-parallelizable sections.

Problem: How to compute a software application as fast as possible while maximizing the time a system can operate in one of the available low-power modes?

Realization: In order to apply the Race-To-Sleep design pattern, the logic and structure of software applications may have to be adapted to monitor and control the CPU frequency during runtime or to divide and allocate the workload to different CPU cores. For single-core environments, frequency alteration must be supported by the operating system or implemented using software libraries and advanced algorithms. Measuring the current workload or performance

counters of the CPU may be used in the decision-making process to adjust the frequency. In multi-core environments, *fork-join* concepts may be added to the software application to share the workload and speed up computation.

Impact on Non-functional Requirements

The power-timing diagram in Figure 4.10 shows the exemplary usage of the *Race-To-Sleep* design pattern in a dual-core scenario.

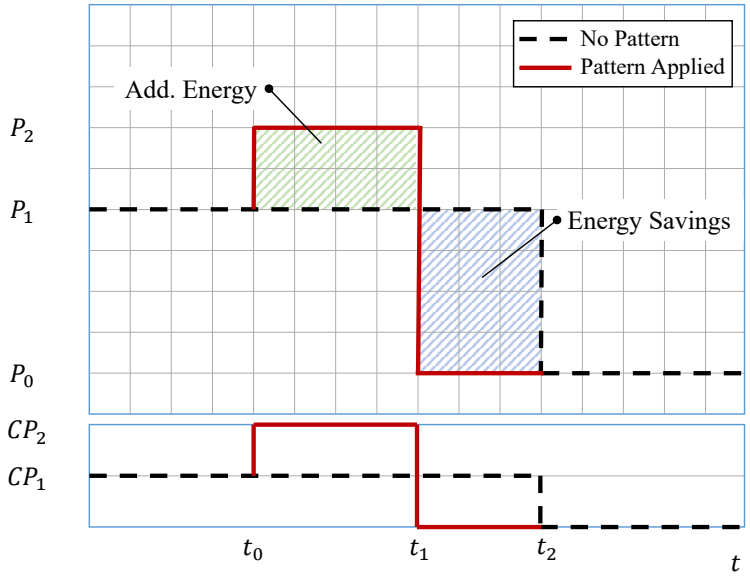


Figure 4.10: Power-timing diagram of the *Race-To-Sleep* design pattern in a dual-core scenario (published in [337]).

Energy Consumption: The upper part of the power-timing diagram shown in Figure 4.10 pictures the power-related temporal behavior of the *Race-To-Sleep* design pattern with the power consumption levels P_0 for the low-power or sleep mode, P_1 for the normal mode, and P_2 for the race mode. In this example, only one CPU core is active in P_1, while both cores are utilized in P_2. In single-core scenarios, P_2 defines the mode with the maximum frequency.

In the lower part of the power-timing diagram shown in Figure 4.10, CP_1 describes the computational power for the normal mode and CP_2 the computational power for the race mode, e.g., both cores of the CPU utilized. The time between t_0 and t_1 represents the active time of the race mode, while t_2 defines the beginning of the low-power mode. With the *Race-To-Sleep* design pattern applied, the software application enters the race mode at t_0, increasing both the computational power and power consumption. After work has been computed at t_1, the software application switches back to the low power mode P_0. To calculate the effectiveness of the *Race-To-Sleep* design pattern, the following definitions and assumptions are required:

$$\Delta t_{10} = t_1 - t_0, \quad \Delta t_{21} = t_2 - t_1 \tag{4.28}$$

$$\Delta P_{10} = P_1 - P_0, \quad \Delta P_{21} = P_2 - P_1 \tag{4.29}$$

which may be used to define the additional energy overhead (E_{add}) and energy savings (E_{save}) of the *Race-To-Sleep* design pattern by means of the following equations:

$$E_{save} = \Delta P_{10} \cdot \Delta t_{21} \tag{4.30}$$
$$E_{add} = \Delta P_{21} \cdot \Delta t_{10} \tag{4.31}$$

Equations (4.28) and (4.29) are used to define the parameters q_P and q_t as follows:

$$\frac{\Delta P_{21}}{\Delta P_{10}} = q_P \tag{4.32}$$
$$\frac{\Delta t_{10}}{\Delta t_{21}} = q_t \tag{4.33}$$

with q_P as the quotient between the additional power consumption in race mode P_2 compared to the power consumption of the normal mode P_1. The quotient q_t describes the ratio between the duration of the race mode and sleep mode when the design pattern has been applied. The energy balance EB_P can be calculated using Equations (4.28) and (4.29) as follows:

$$\begin{aligned} EB_P &= E_{save} - E_{add} \\ &= \Delta P_{10} \cdot \Delta t_{21} - \Delta P_{21} \cdot \Delta t_{10} \\ &= \Delta P_{10} \cdot \Delta t_{21}(1 - q_P \cdot q_t) \end{aligned} \tag{4.34}$$

The value of EB_P is positive if the energy savings are larger than the energy overhead introduced by the additional core to finish the computation earlier. The efficiency factor η_P can be defined as $(1 - q_P \cdot q_t)$. The calculations of EB_P and η_P can be evaluated with the help of Amdahl's law [297], which describes the execution time of an application when switching from a sequential to a parallel approach. The speedup S described in Amdahl's law is defined as:

$$S = \frac{T_S}{T_P} = \frac{1}{f_s + \frac{1-f_s}{p}} \tag{4.35}$$

T_S represents the sequential execution time, f_s is the sequentially performed proportion of an algorithm with $0 \leq f_s \leq 1$, p defines the number of CPU cores used, and T_P defines the parallel execution time with $T_P = f_s \cdot T_S + ((1 - f_s)/p) \cdot T_S$. For the evaluation, the following algebraic relations are introduced:

$$\Delta t_{20} = T_S \tag{4.36}$$
$$\Delta t_{10} = T_P \tag{4.37}$$
$$\frac{\Delta P_{21}}{\Delta P_{10}} = g \cdot (p - 1) = q_P \tag{4.38}$$
$$\frac{\Delta t_{10}}{\Delta t_{21}} = \frac{T_P}{T_S - T_P} = \frac{1}{S - 1} = q_t \tag{4.39}$$

In Equation (4.38), $g = [0, 1]$ is defined as the relative proportion of the power consumption ratio q_P for an additional single core, and $(p - 1)$ defines the number of additional cores. With

Equations (4.36) to (4.39), the definitions of EB_P and η_P may be verified as follows:

$$\begin{aligned} EB_P &= (\Delta P_{10} \cdot (T_S - T_P)) - (\Delta P_{21} \cdot T_P) \\ &= (\Delta P_{10} \cdot (T_S - T_P)) - (\Delta P_{10} \cdot g \cdot (p-1) \cdot T_P) \\ &= \Delta P_{10} \cdot (T_S - T_P) \left(1 - q_P \cdot \frac{T_P}{T_S - T_P}\right) \\ &= \Delta P_{10} \cdot \Delta t_{21} (1 - q_P \cdot q_t) \end{aligned} \tag{4.40}$$

and

$$\eta_P = 1 - g \cdot (p-1) \cdot \frac{1}{S-1} = 1 - q_P \cdot q_t \tag{4.41}$$

Considering the MPC8641D dual-core CPU [267, 374] as an example. The additional energy consumption is 30 % ($g = 0.3$) higher compared to a single-core usage. With an overhead of 10 % (non-parallelizable part of the application) $f_s = 0.1$ and $p = 2$ for the dual-core setup, the following calculations for η_P can be performed:

$$S = \frac{1}{0.1 + 0.45} \approx 1.\overline{81} \implies q_t \approx 1.2, \;\; q_p = 0.3 \tag{4.42}$$

$$\eta_P = 1 - (0.3 \cdot 1.2) = 0.64 \tag{4.43}$$

Execution Time / Latency: The *Race-To-Sleep* design pattern may accelerate the execution of the software application and has a positive effect on the overall time behavior. However, if power consumption levels change dynamically at runtime, the execution time of a software application is difficult to predict.

Consequences: Developers have to consider other peripheral devices, such as timers, when adjusting the frequency of a CPU since they may use the same clock generators. Clock-rate changes can also lead to negative side effects and undefined behavior when timer-related intervals are used. Moreover, the application has to be designed without blocking accesses and waiting periods. Parts that cannot be parallelized may reduce the overall efficiency of the design pattern.

Chapter 5

Power Estimation Concept for MDD

Energy-aware design patterns to enhance software application during the architecture and design phases have been presented in Chapter 4 (p. 4 ff.). This chapter focuses on concepts required to implement the developer workflow presented in Section 3.1 (p. 77 ff.) for the power consumption estimation of software application models as contributions to answer RQ3.

Section 5.1 provides a brief overview of the vision for the power consumption estimation concept presented in this thesis. The remaining sections provide contributions to address RQ3 and parts of RQ4 defined in Section 1.2.2 (p. 8 ff.). Section 5.2 discusses the characteristics of hardware components essential for the hardware modeling process and introduces hardware component models along with their realization in UML. The UML-based *Power Analysis Profile (PAP)* to model power-related characteristics of hardware components is described in Section 5.3. Finally, two methods for the power consumption estimation are introduced in Section 5.4, which can be applied in different MDD phases. Initial ideas for this approach have been published in [339, 340, 341].

5.1 Overview

In this section, the motivation and the overall goals for the power consumption estimation concept are discussed. As shown in Figure 5.1, the relations between the concepts and methods of this chapter can be illustrated as a three-step process.

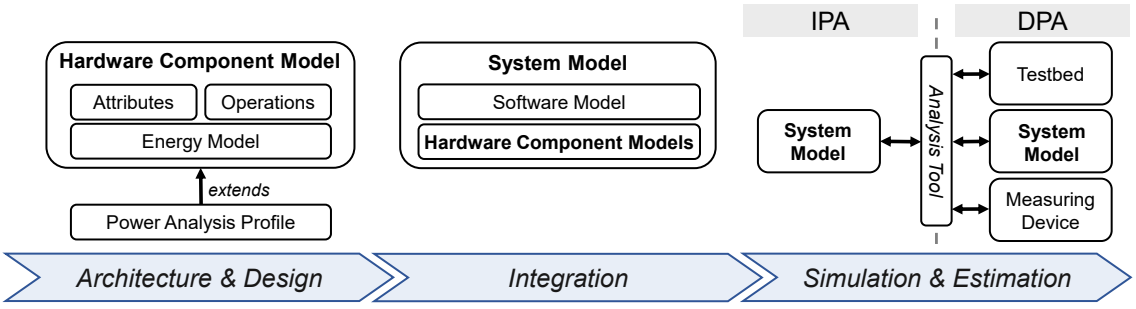


Figure 5.1: Overview and relations between the concepts and methods for a power consumption estimation. Horizontal arrows describe bi-directional communication between components, while the vertical arrow describes a concept extension.

In the first step, the concept discussed in Section 5.2 (p. 117) is used at the architecture and design phase to specify hardware component models as proxies of physical hardware components in UML with structural aspects such as attributes and operations and an energy model describing discrete energy-related behavior.

The PAP introduced in Section 5.3 (p. 124 ff.) extends hardware component models to allow engineers and developers to specify energy- and time-related NFP. In the second step, hardware component models are integrated into the UML-based software application model to make hardware-software interactions visible and to form a system model. The system model is simulated in the third step for a power consumption estimation. The simulation is based on one of two power analysis methods introduced in Section 5.4 (p. 135 ff.). The *Indirect Power Analysis (IPA)* method does not require additional hardware, while the *Direct Power Analysis (DPA)* method utilizes a physical testbed and enables direct interaction between the simulated system model and physical hardware function-wise and energy-wise. A power consumption estimation by both methods results in an energy trace. The energy trace is used to analyze misbehavior and energy bugs in software applications based on predefined NFRs.

As mentioned in Section 3.1 (p. 77 ff.), the concepts and methods of this chapter are integrated into various steps of the developer workflow. Software engineers may use hardware component models and the PAP during the architecture and design phase (cf. actions 3(a) to 5 in Figure 3.1, p. 78) to build the system model. Test engineers may use the provided power analysis methods during simulation to perform power consumption estimations (cf. right side of Figure 3.1, p. 78). In summary, the power consumption estimation concept for MDD intends to achieve the following goals:

- Providing hardware component models which may be integrated into a software application model to make hardware-software interactions visible.

- Definition of a UML profile to extend hardware component models with power-related NFPs as a basis for the power consumption estimation process.

- Enable interaction between the system model and real hardware components during the simulation to provide true data, e.g., from sensors, and to avoid the use of simulated data. This also enables more advanced simulations, which can consider the entire communication process, e.g., from an IoT node through a gateway to a cloud service.

- An evaluation based on real power consumption measurements that can be mapped to the behavior of the software application model. The impact of software model changes on power consumption can be derived directly. The complete development cycle is shortened and is suitable for rapid prototyping approaches.

- A decoupling between the software model and the test environment by the use of a *Model-Testbed* as a special execution platform that supports the exchange of individual components, e.g., to consider different variations of the same sensor type and the evaluation of the impact on the overall power consumption of the system.

- A more energy-transparent software application model [135] due to the detailed simulation and analysis approach to support energy-aware decisions based on the evaluation of power-related NFRs.

5.2 Hardware Modeling

This section covers the hardware modeling process and the development of hardware component models. Section 5.2.1 discusses the power-related characteristics of hardware components. Section 5.2.2 provides a definition of hardware component models, while Section 5.2.3 presents their implementation in UML to enable a linkage with the software application model.

5.2.1 Characteristics

As mentioned in Section 2.2.2 (p. 24 ff.), the software level and hardware level have a cause-effect relationship [160] where actions of the software application are causing effects on hardware components resulting in specific power consumption of hardware components. Therefore, it is essential to consider the underlying hardware for a proper power consumption estimation of software applications. Based on the aspects of hardware components, they can be classified according to different characteristics. A classification based on the energy management capabilities provided by hardware components is presented below. In addition, their energy behavior is particularly interesting for the modeling process. As further discussed in this section, the behavior of hardware components can be described with a black-box, grey-box, or white-box transparency, directly affecting the modeling accuracy.

Introduction of Device Classes

Due to the variety of MCUs and peripheral device types, defining a general abstraction of the functionality and properties of these devices is a major challenge. Hence, the concept of device classes is introduced as a first step towards a classification for the hardware modeling process. The basic idea of device classes has been published in [391] as a follow-up work of [337] and divides hardware components into one of the following four device classes based on their power management capabilities:

- *Class 0*: Devices that do not offer any possibilities to regulate power consumption by design. The only option for the software application to gain control over such devices to turn them on and off externally is to use additional hardware components, such as a transistor or programmable power supply. However, if these components are not already part of the embedded system, a rework of the circuit design is required.
- *Class 1*: Devices without a discrete power management unit but with the functionality to control the operating mode via the software layer. Typically, such devices provide at least two power states for an active and a power-off mode.
- *Class 2*: Devices with their own internal logic realized in hardware and the firmware to control and adjust the power consumption. Typically, the selected power state affects the functionality and quality of service (e.g., precision, number of measurements) of the device. On the software level, it may be possible to switch between different power states. Note that devices of this class may not have the ability to be turned off completely.
- *Class 3*: Devices with full power management capabilities that support all previous *Class 1* and *2* criteria. The combination of proprietary logic and the ability to control the device externally and entirely through the software application enables the implementation of sophisticated power management strategies.

Generally, possible power optimization approaches depend on the complexity of the considered device. From a technical perspective, *Class 0* devices, such as low-complex sensors, may only be optimized externally by adjusting the voltage and electric current as described by Equation (2.1) in Section 2.1.1 (p. 17 ff.). Such devices are also denoted as simple devices in [337]. In contrast, the power consumption of *Class 1–3* devices, denoted as complex due to their internal architecture and dynamic behavior, may be expressed using Equations (2.2) to (2.5). Power optimization approaches for complex devices on the software level (cf. Section 2.2.2, p. 24 ff.) mainly focus on dynamic aspects, including voltage, electric current, and frequency.

Black-box, Grey-box and White-box Transparency

In addition to the power management capabilities, another important characteristic is the behavior transparency of a hardware component, which can be classified as black-box, grey-box, or white-box transparency, a concept similar to the dynamic testing techniques introduced in Section 2.7.2 (p. 53 ff.). In particular, *Class 1–3* devices often have a complex behavior that results in multiple operating states with different power consumption levels. In the following, the two main challenges related to the acquisition of information for the hardware modeling process are discussed, whose complexity depends on the transparency of the hardware component to be modeled.

- The first challenge concerns the identification of existing operating states for the hardware modeling process. Typically, data sheets provided by manufacturers contain a detailed description of defined operating states and their characteristics, e.g., power consumption or operating frequencies. However, a hardware device may have hidden states that can be easily overlooked during modeling, e.g., if they are undocumented or only active when specific events occur. The detection of such hidden states requires a more comprehensive evaluation, which may be achieved by approaches as described in [127]. However, since hardware component models in this thesis focus on non-functional aspects, operating states may be combined, from an energy perspective, into a single state if their functional behavior has identical power consumption or timing characteristics.

- The second challenge concerns the identification of transitions between operating states and their representation in hardware component models. In total, three different types have to be considered. A state change of a hardware component may be triggered (a) by the software application, (b) by the hardware component itself (intrinsic), and (c) by the environment as an extrinsic force. A wireless communication device, for instance, will enter a transmission state after the software application sends a transmit command (a) for a specific message. Due to their internal logic, hardware components can initiate state transitions independently (b), for instance, open receive windows or switch to a low-power mode after a timeout. The last type of state change (c) is initiated by the hardware component due to environmental changes. With simulations focused on the software application, the environmental impact cannot be covered without extending the simulation environment to include environmental models. The characteristics of the environment may be modeled by using the concepts of scenarios introduced in Section 3.2 (p. 81 ff.). Although the methods for the simulation and analysis presented in Section 5.4 are suitable for taking the impact of the environment into account in a simplified manner, the extension of simulation environments is not the main focus of this thesis and is considered as out of scope (cf. Section 1.3, p. 12 f.).

Imagine the Plantower PMSA003I laser-based particle concentration sensor [419] as an example. The sensor adjusts the sampling mode from a so-called *normal mode* to a *fast mode* automatically due to the particle density in the environment, which refers to a type (c) transition. In addition, the threshold for switching between modes is not documented. Since the Plantower PMSA003I does not provide an interface to retrieve the active operating state, the *fast mode* can be considered a hidden state that only becomes visible by observing a change in the overall power consumption of the sensor.

In summary, the behavior of a hardware component may be considered as white-box behavior if all operating states are known and are part of the hardware component model and all transitions are initiated by the software application during simulation. A grey-box behavior is similar to a white-box behavior, but transitions may also be initiated by the hardware device itself as intrinsic behavior, which may have to be predicted by the hardware component model during simulation. The behavior of a hardware component can (partly) be considered as black box behavior if hidden states exist that are not covered by the hardware component model. Additionally, transitions are triggered by environmental changes, which may not be simulated with state-of-the-art simulation environments of current MDD tools (cf. Appendix A.2, p. 265 ff.).

5.2.2 Formal Definition of Hardware Component Models

This section presents the basic concept for the formal abstraction of hardware components. Before discussing the concept of hardware component models, the concept of energy models as a fundamental element of hardware component models and the characterization of events as an important part of modeling reactive systems are introduced.

Concept of Energy Models

Energy models describe the non-functional energy-related behavior of hardware components. In the following, the term SUT_{ES} refers to an embedded system used as a system under test, which contains a finite set of n independent hardware components C_n, defined as:

$$\mathrm{SUT}_{ES} = \{C_1, C_2, ..., C_n\} \tag{5.1}$$

The behavior of each hardware component $C_i \in \mathrm{SUT}_{ES}$ may be expressed using a *Finite State Machine (FSM)* [410]. We define the FSM for the description of power-related behavior as the tuple $(\Sigma, S_p, s_0, \delta, V_c, I_s, I_\delta, T_s, T_t)$ with the following elements:

- Σ, as the input alphabet.
- $S_p : |S_p| \geq 2$, as a finite set of power states.
- $s_0 \in S_p$, as the initial power state.
- $\delta : S_p \times \Sigma \rightarrow S_p$, as the state-transition function.
- V_c, as the supply voltage of the hardware component in volt (V).
- $I_s : S_p \rightarrow \mathbb{R}_{\geq 0}$, as the electric current consumption of a power state in ampere (I).
- $I_\delta : S_p \times \Sigma \rightarrow \mathbb{R}_{\geq 0}$, as the electric current consumption of a transition in ampere (I).

- $T_s : S_p \times \Sigma \to \mathbb{R}_{\geq 0}$, as the execution time of a state in milliseconds (ms).
- $T_t : S_p \times \Sigma \to \mathbb{R}_{\geq 0}$, as the execution time of a transition in milliseconds (ms).

Each state $s \in S_p$ may refer to a functional operating mode of a hardware component, e.g., $\{On, Off, Idle\}$. A state change from state $s1 \in S_p$ to state $s2 \in S_p$ can be achieved by executing the transition $t_{12} \in \delta$, triggered by an event $e_{12} \in \Sigma$, expressed as $s1 \xrightarrow{t_{12}|e_{12}} s2$.

For simplicity, the supply voltage V_c will be considered static for the remainder of this thesis, while concepts such as voltage scaling [300] are considered out of scope. However, the proposed definition may be extended to include a list of discrete voltage levels, e.g., $V_c = \{0.0, 2.0, 3.3, 5.0\}$, where a specific voltage level $v \in V_c$ may be constant across all states and transitions for a specific time t. A voltage level may also be mapped to specific states or transitions, leading to the following adaptations of previous definitions:

- V_c, as a set of supported voltage levels of the hardware component in volt (V).
- $I_s : S_p \times V_c \to \mathbb{R}_{\geq 0}$
- $I_\delta : S_p \times \Sigma \times V_c \to \mathbb{R}_{\geq 0}$

Since each state and transition of a hardware component is annotated with power and timing aspects, energy models are suitable for estimating the electric power consumption P, which may be calculated for a specific point in time using the supply voltage V_c and the time-dependent current consumption of either the active state ($I_s(t)$) or the active non-instantaneous transition ($I_\delta(t)$), cf. Section 2.1.1 (p. 17 ff.). Based on the energy models of n hardware components and Equations (2.1) to (2.7), the power consumption P_{es} and the energy consumption E_{es} of an embedded system may be estimated as follows:

$$P_{es}(t) = \sum_{i=1}^{n} P_{C_i}(t) \tag{5.2}$$

$$E_{es}(T) = \int_{t=0}^{T} P_{se}(t)\, dt = \int_{t=0}^{T} \left(\sum_{i=1}^{n} P_{C_i}(t) \right) \tag{5.3}$$

An Energy model may not map all logical or functional operating modes of a hardware component. For instance, two operating modes can be aggregated into a single state if they have the same impact on the overall power consumption or if they cannot be externally observed or distinguished, which is described as black-box behavior in Section 5.2.1 (p. 117 ff.). Vice versa, an operating mode of a hardware component can be separated into different power states if the operating mode has a distinguishable power consumption for each phase, e.g., if a state of a sensor includes a preheat and a data acquisition phase.

Characterization of Events

The input alphabet Σ of an energy model defines a list of events that trigger state transitions. Considering the sources for state changes as part of the hardware characteristics discussed in Section 5.2.1 (p. 117), a distinction can be made between the following three types of events:

- Application-triggered: Describes all types of events caused by the behavior of the software application model. Those events are declared as extrinsic events. For example, enabling and disabling peripheral devices or initiating a wireless network transmission may be triggered by this type of event.

- Time-triggered: Describes all types of intrinsic events that are generated by the device itself and occur at a constant or predictable time after the state has been entered, for example, if the execution time of a state has been specified.

- Interrupt-triggered: This event type is defined as spontaneous intrinsic. This type of event is generated by the device itself, e.g., due to environmental changes as intrinsic forces. For the test case definitions, scenarios (cf. Section 3.2, p. 81 ff.) may be used to define such events and points in time for their appearance and may be generated by an environmental model during the simulation. As stated in Section 5.2.1 (p. 117 ff.), interrupt-triggered events are part of the black-box behavior transparency not covered by state-of-the-art simulation environments of current MDD tools and considered as out of scope for this thesis.

While state transitions δ have a fixed execution time and are assumed to be uninterruptible, the execution times of states $s \in S$ may not be predictable in advance, e.g., if state transitions depend on application- or interrupt-triggered events defined by the input alphabet Σ and, thus, on the behavior of the software application or environmental model.

Concept of Hardware Component Models

The dynamic behavior of embedded systems with varying power consumption levels has been analyzed, e.g., in [239, 427]. According to [42, 426], each hardware component may be described with a set of states defining functional operating modes and transitions to switch between those modes. The concept referred to as power state machine [42, 87] extends the description of functional modes and transitions to include meta-information about power consumption and execution time. By modeling supply voltage, electric current consumption, and timing aspects for states and transitions, the concept of energy models is able to consider the impact on NFPs in detail. Furthermore, since some aspects introduced by energy models, such as I_s as the electric current consumption of a state, may be defined as variable over time, dynamic behavior can be considered more accurately when modeling hardware components, which is discussed in-depth in Section 5.3 (p. 124 ff.). In this thesis, the abstract model of a hardware component is formally defined by the tuple $(EM_{hc}, O_{hc}, A_{hc})$, with the elements:

- EM_{hc}: Energy model of the hardware component with a list of power states (operating modes), a list of state transitions, and power-related characteristics.

- A_{hc}: Finite set of attributes to model the inner state of the hardware component, e.g., the active configuration.

- O_{hc}: Finite set of operations as an interface for software models, e.g., to change the configuration and generate events.

EM_{hc} corresponds to the previously introduced energy model as a key element containing all power states, transitions, and events to model the power-related behavior of a hardware component. To achieve a more accurate and realistic modeling approach, existing concepts discussed in Section 2.8 (p. 65 ff.) are extended by including dynamic power characteristics in states and transitions. Instead of defining static values, the NFPs of states and transitions may vary during simulation. For the calculation of NFP values at specific points in time, e.g., the power consumption for a state, the configuration of a hardware component is stored in attributes of

A_{hc}. Imagine a hardware component model of a sensor device with $S_p = \{On, Off, Measure\}$ as an example. The software application controls the sensor with application-triggered events initiated by operations to turn the sensor on and off and to start measurements. Due to this, the retention times for the states *On* and *Off* are not predictable in advance and depend on the behavior of the software application during execution. Suppose a measurement is initiated (application-triggered event). In this case, the execution time of the measurement, e.g., the time the hardware component model stays in the *Measure* state, may be calculated based on the parameters in A_{hc}. The transition between the states *Measure* and *On* may be triggered internally by a time-triggered event after completion.

5.2.3 Integration into Software Models

For the overall concept introduced in Section 3.1 (p. 77 ff.), the integration of hardware component models into the software model domain is an important step for the specification of a system model used to estimate power consumption and detect energy bugs. Although SysML focuses more on modeling systems and hardware components (cf. Section 2.5.1, p. 38 ff.), hardware component models can be mapped directly to different UML model elements. By using UML as a modeling language for both system aspects, hardware component models can be completely integrated into the software domain which addresses RQ3.

For the representation of energy models, this thesis uses UML state machine diagrams. The underlying concept of UML state machines extends the mathematical model of FSMs and is therefore able to fully represent the concept of energy models. UML provides additional diagrams to describe the dynamic behavior of objects over time. However, these diagram types have several disadvantages, which are described in the following:

- As a behavioral diagram, the UML sequence diagram focuses on time-based interactions between objects. Since the modeled behavior should also be reflected in state machines describing the behavior of objects, both diagrams share a close relation [144]. By modeling a sequence of messages between objects, the execution of a particular use case is described. In contrast, state machine diagrams, are used to specify the inner behavior of an object across several use cases by defining what actions can be executed depending on the current state of the object. By this, state machine diagrams are more suitable to represent energy models.

- The UML activity diagram is a behavior diagram for the description of workflows using activities and actions while focusing on concurrency and synchronization. While the UML state machine diagram describes the states of an object and possible transitions between the states as reactions to events, activity diagrams describe a flow of activities executed one after another without the need for events or triggers. Since the energy-related behavior of hardware components depends on events, activity diagrams are less suitable.

UML classes model structural and static aspects of hardware components, such as attributes and operations. The UML specification also provides component diagrams to specific structural aspects at a higher level of abstraction. Each component in a component diagram is described with its ports, required and realized interfaces defining the behavior, and relationships to other components.

However, component diagrams have several disadvantages:

- Even though components may be extended with state machines, UML component diagrams can not be simulated in MDD tools such as IBM Rhapsody.
- UML ports of component diagrams specify the required and requested interfaces that also may contain operations. However, they do not have a direct mapping to concepts of object-oriented programming languages like C++, leading to a MDD tool-specific code generation that needs to be adapted manually in later development phases.
- During the automatic code generation process, components are realized as classes. For each class within a component, an inner class is generated, leading to complex and unmaintainable source code. In IBM Rhapsody, the source code generation for component diagrams is also limited [168].

Since UML class diagrams are more well-known and frequently used by developers than other structural diagrams [97, 320], they are selected to model the static aspects of hardware component models. Moreover, associations and aggregations instead of ports are used to define relations due to the simpler implementation. In the UML metamodel, a UML class is specified as a UML classifier and specializes the abstract classes `BehavioredClassifier` and `EncapsulatedClassifier` [275]. As a result, a UML class has associated features representing structural and behavioral characteristics, e.g., attributes and operations. In addition, the concept of energy models may be mapped directly to UML behavioral state machines, which, in turn, can be assigned to the *classifierBehavior* property of a UML class to specify the behavior a UML class exhibits over time as a direct consequence of actions.

From an implementation point of view, the general idea of hardware component models follows the hardware proxy pattern [99], in which a proxy class is responsible for a specific hardware device and encapsulates all accesses to the device. Additionally, the proxy class may contain public and private functions and a set of attributes. However, the presented concept of hardware component models extends the proposed pattern in [99] by the additional consideration of non-functional aspects due to the energy model. In addition to operations that invoke functional behavior, a class representing a hardware component model also provides functions that directly affect power-related behavior by generating events impacting the associated energy model, such as turning a hardware component on and off. As a basis for the development, this thesis uses the two exemplary abstract classes shown in Figure 5.2 for the initial derivation of hardware component models. Intended as an orientation for software developers, one may extend this basic idea of abstract classes to add or remove operations or provide further classification without affecting the overall modeling approach.

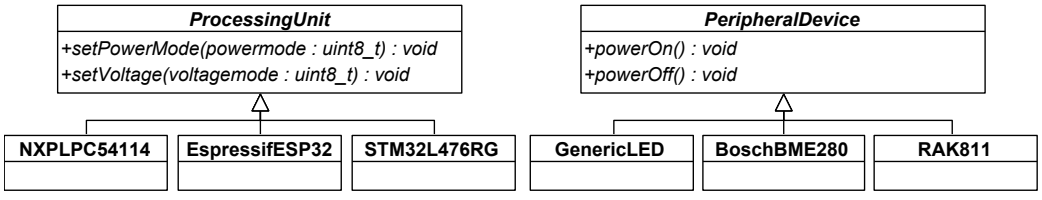


Figure 5.2: Abstract classes with basic power-related functions to define hardware component models, illustrated as UML 2.5 class diagram (published in [338]).

All hardware component models are derived from one of the two base classes, `PeripheralDevice` and `ProcessingUnit` (cf. Figure 5.2), providing device-specific basic power-related operations as a basic interface for software applications. The `PeripheralDevice` class is used, e.g., for sensors, actors, and communication interfaces. In order to extend the provided interface and make specific functionalities of a hardware component accessible, existing driver descriptions may be used as a source for function signatures, data types, and configuration parameters. If MDD tools generate source code in later steps (cf. Figure 2.21, p. 59), hardware component models may directly be replaced with existing driver implementations due to their similar function signatures while the generated source code of the software application model does not require additional manual adjustments by software developers.

The class `ProcessingUnit` is a base class for hardware component models describing MCUs and provides methods for their configuration and for switching between operating modes. However, MCUs differ significantly from each other and must therefore be considered individually. Furthermore, hardware component models for MCUs require further abstraction since they are more challenging when it comes to simulations in later MDD phases. Unlike peripheral devices, changing the configuration or power mode of an MCU directly affects the life cycle of the software application itself. To keep the software application model platform-independent, a specific HAL for MCUs has to be provided, considering different operating modes in an abstract manner. As part of the prototype implementation, Section 6.5.2 (p. 163 ff.) addresses the topic of abstraction more in-depth.

5.3 Power Analysis Profile (PAP)

To implement the concepts of hardware component models and energy models in UML, this section introduces the PAP profile as the key approach to model non-functional aspects of hardware components and to provide quantitative values for the analysis process. The PAP is based on MARTE and enables the assignment of power- and energy-related properties for UML-based state machines and class definitions.

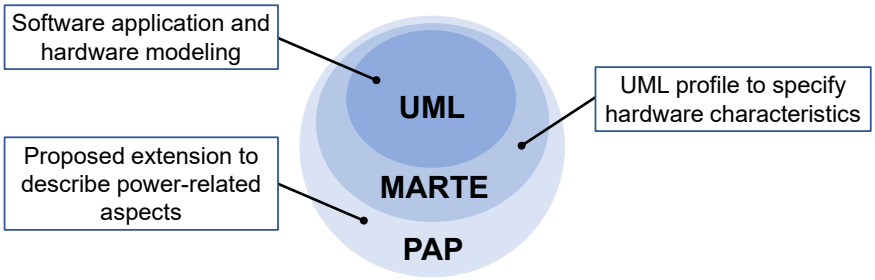


Figure 5.3: Relations between the UML, MARTE, and the PAP, illustrated as Euler diagram.

Figure 5.3 shows the relation between the subset of UML used to define software and hardware component models (cf. Section 5.2.3, p. 122 ff.), MARTE as an extension of UML (cf. Section 2.6, p. 41 ff.), and the PAP to model power- and timing-related aspects based on MARTE. The PAP is a contribution to answer RQ3 and is conceptually located in step 4 of the developer workflow, shown in Figure 3.1 (p. 78).

The reason for defining a UML profile as a collection of stereotypes to model NFPs of hardware components is motivated as follows:

- The description of NFPs with UML comments allows the element-wise annotation of states, transitions, attributes, or operations. As notes in diagrams, UML comments can also be represented graphically. However, they do not provide a predefined structure, and developers can mix-up arbitrary dimensions and information, which makes them not directly machine-readable. For the automatic processing of NFPs, a semantic description and a corresponding parser have to be developed.

- The NFPs of a hardware component model may be specified as attributes along with the functional aspects within a UML class diagram, e.g., by defining a proper naming scheme. As a result, NFPs are also represented in model-to-text transformations (code generation), even though they have no purpose at the functional level. To avoid this, an adapted model transformation process must be defined to remove attributes representing NFPs from the model prior to code generation, which unnecessarily complicates the work of software developers and can make modeling confusing. Since runtime information and the internal state of a hardware component model instance (object) are required to calculate metrics related to the dynamic energy-related behavior, NFPs are instead annotated with specific stereotypes at the meta-level.

UML stereotypes provide a standardized concept to extend the UML metamodel to add or redefine the meaning of UML elements. For instance, MARTE and SysML heavily rely on the use of stereotypes for their specifications. Structured as key-value pairs, stereotypes also provide a well-defined, type-safe, and machine-readable structure, e.g., when accessed via APIs of MDD tools. Additional background on stereotypes can be found in Appendix A.3 (p. 266 ff.).

The following Section 5.3.1 gives a brief overview of the basic structure of the PAP, while Section 5.3.2 describes the extension of MARTE [278] as a basis for the definition of the new PAP UML profile. Sections 5.3.3 to 5.3.4 describe the packages of the PAP in depth. The concept of modeling dynamic power-related behavior is explained in Section 5.3.5. Additionally, a hardware component model of a dimmable LED is presented as an ongoing example in Sections 5.3.3 to 5.3.5 to illustrate the usage of the concept.

5.3.1 Overview

The mapping of the energy model introduced in Section 5.2.2 into UML is based on the profile extension mechanism (cf. Appendix A.3.2, p. 270 ff.) to extend the UML metamodel with stereotypes containing the semantic description of domain-specific information, e.g., power-related aspects. All stereotypes introduced by the PAP are thematically assigned to one of the two sub-profile packages shown in Figure 5.4 based on their particular purpose and used to specify power-related aspects of hardware component models. Stereotypes for structural aspects of hardware component models are described in Section 5.3.3, while stereotypes for UML behavioral state machines as the basis of energy models are introduced in Section 5.3.4. The basic definition of the PAP has been published in [338] and modified and expanded in [341]. It has been adapted again for this thesis due to the most recent findings. Since the presented profile is based on MARTE [278], the following Section 5.3.2 introduces the extension of the MARTE profile before presenting the stereotypes of the sub-profile packages in detail.

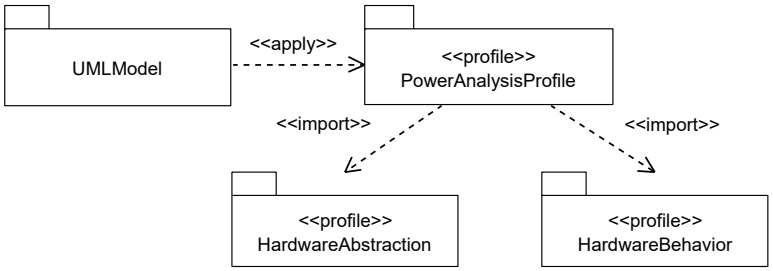


Figure 5.4: Overview of PAP profiles and sub-profiles (UML 2.5 package diagram notation).

5.3.2 MARTE Extension

The MARTE library (cf. Section 2.6, p. 41 ff.) consists of a number of basic data types (*NFP_Types*) to describe NFPs and measurement units as parts of *NFP_Types* that specify units and conversion rules. As shown in Figure 2.14 (p. 48), the MARTE library provides *NFP_Types* with associated measurement units to model power and energy. However, for the overall concept of this thesis, a more detailed modeling of electric current and voltage is required, which is not provided by the MARTE specification. For this, the MARTE library is extended by additional data types. Based on the notation provided by the MARTE specification [278, 348], Figure 5.5 shows the definitions of the newly defined basic *NFP_Types* and their corresponding measurement units.

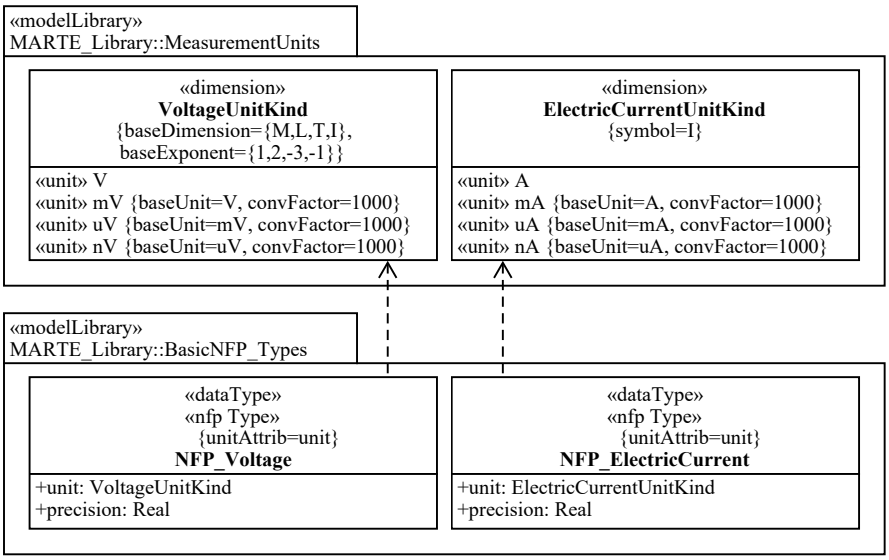


Figure 5.5: Additional data types for the MARTE library to describe voltage and electric current, adapted from [338] (UML 2.5 profile diagram notation).

The *ElectricCurrentUnitKind* measurement unit at the top right of Figure 5.5 represents the base *International System of Units (SI)* metric for electric current with the physical base dimension I, as explained in Section 2.1 (p. 17 ff.). The tags `baseUnit` and `convFactor` in the upper part of Figure 5.5 specify the unit of the type and the conversion rules within the same unit. As a derived SI metric, the *VoltageUnitKind* consists of the base dimensions for mass (M),

length (L), time (T), and electric current (I). With `NFP_Voltage` and `NFP_ElectricCurrent` in the lower part of Figure 5.5, additional data types for the measurement units are defined and added to the *NFP_Types* section of the MARTE library. Those *NFP_Types* are used to model non-functional aspects of hardware component models, such as electric current consumption, in a dynamic and detailed manner.

5.3.3 Hardware Abstraction Package

The sub-profile *HardwareAbstraction* of the PAP (cf. Figure 5.4, p. 126) provides stereotypes to extend UML classes and UML class elements of hardware component models with additional information primarily used for the simulation and analysis described in Section 5.4. Figure 5.6 shows an overview of the stereotypes provided by the *HardwareAbstraction* sub-profile of the PAP. The stereotype `HwAbstraction` can be applied to UML classes and is primarily used

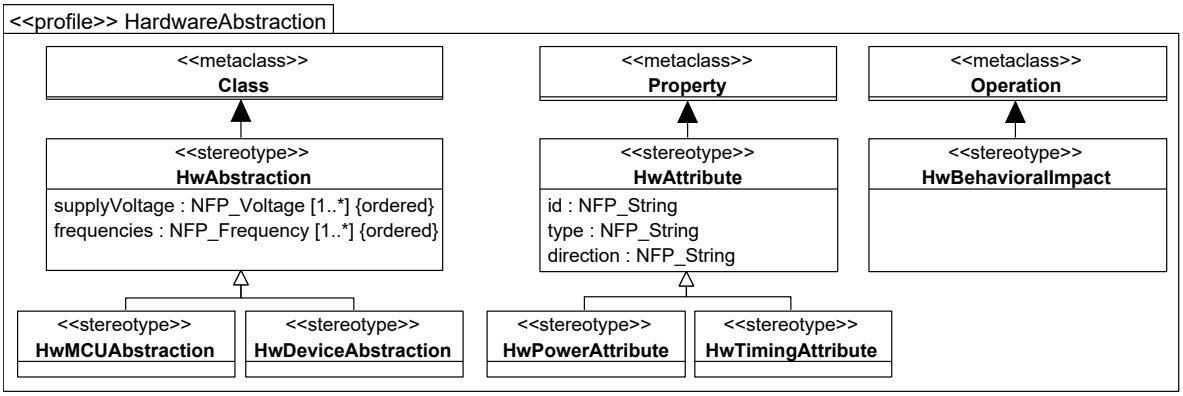


Figure 5.6: *HardwareAbstraction* sub-profile package, adapted from [341] (UML 2.5 profile diagram notation). Generalization hierarchies and unused tags have been omitted.

to tag hardware component models within a system model. By this, developers are able to distinguish between classes of the hardware component model and classes of the software application model. Additionally, model transformation processes, as explained in the developer workflow described in Section 3.1 (p. 77 ff.) and Section 5.4 (p. 135 ff.), rely on such markers for their automatism. Besides the use as a marker, the `HwAbstraction` stereotype provides tags for basic properties (cf. Figure 5.6), such as supported frequencies and the definitions of the supported supply voltages expressed as a tagged value based on the introduced `NFP_Voltage` data type. The stereotypes `HwMCUAbstraction` and `HwDeviceAbstraction` share the same set of tags and allow the distinction between processing units and peripheral devices, as described in Section 5.2.3 (p. 122). Note that Figure 5.6 only contains tags used by the current approach. While the tag `supplyVoltage` has been additionally defined, the tag `frequency` is available due to the inheritance of the MARTE stereotype `HwResource` of the HRM package, cf. Section 2.6 (p. 41 ff.) and [278]. For a complete list of tags of the stereotype `HwAbstraction` resulting from generalization hierarchies within MARTE, see Table B.1 in Appendix B (p. 273 f.).

Attributes of a hardware component model affecting power-related aspects, such as electric current consumption or execution time, may be annotated with the `HwPowerAttribute` or `HwTimingAttribute` stereotype, respectively. Both stereotypes inherit from the base stereo-

type `HwAttribute` which provides tags to define a unique *Identifier (id)* as an attribute identifier. By this, a connection between an element on the functional level and an element on the meta-level can be achieved. The mechanism is used by the methods described in Section 5.4 (p. 135 ff.) to include changes in the calculation of the electric current consumption and execution time during runtime. The remaining tags `type` and `direction` describe further properties of attributes annotated with the `HwPowerAttribute` or `HwTimingAttribute` stereotype based on the VSL specification [278], which may be included in model-to-model or model-to-text transformations (cf. Section 2.5.2, (p. 40 ff.). Since `HwAttributes` are typically used in expressions, the `type` is typically set to boolean (0 and 1 for false and true) or numerical data types such as integer, float, or double. The `direction` tag defines whether the variable in the context of an expression is an input (`in`), output (`out`), or both (`inout`).

The stereotype `HwBehavioralImpact` may be applied to any operation that either affects one or more attributes annotated with the `HwPowerAttribute` or `HwTimingAttribute` stereotypes or directly affects the power-related behavior, e.g., by triggering events leading to state changes of the energy model. Similar to the stereotype `HwAbstraction`, the `HwBehavioralImpact` is mainly used as a marker for later model transformation processes. For instance, additional automatic source code generation steps [162] may be performed prior to the simulation and analysis phase to add further trace commands to the opaque behavior, including the logging of affected attributes and their new values.

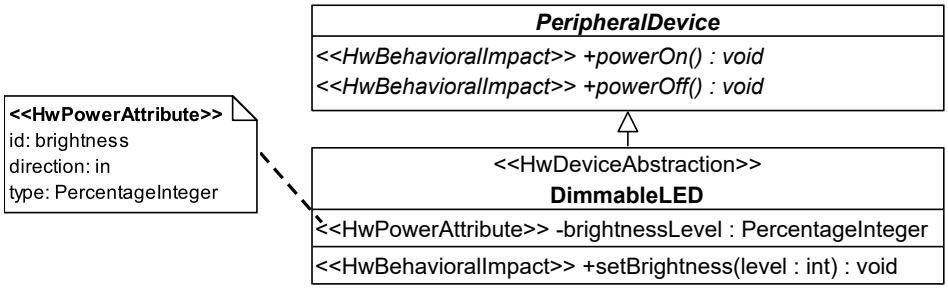


Figure 5.7: Class definition of the dimmable LED example (UML 2.5 class diagram notation). Unused tagged values have been omitted to improve legibility.

Ongoing Example: Structural Modeling

In the following, the structural part of the hardware component model for the ongoing dimmable LED example is introduced to illustrate how developers may use the stereotypes provided by the *HardwareAbstraction* package in MDD. The UML class defining the structure of the hardware component model is shown in Figure 5.7. As specified in Section 5.2.3, the class `DimmableLED` inherits from class `PeripheralDevice` to provide at least a set of two predefined operations so that the software application model can influence the power-related behavior by turning the device on and off. The `DimmableLED` class provides the attribute `brightnessLevel` to represent the current brightness level of the LED in percent, whose value may be changed by the software application using the `setBrightness` operation. The `brightnessLevel` attribute is annotated with the `HwPowerAttribute` stereotype to specify an *id* for the use in expressions.

Following the MARTE specification [278], an own integer-based datatype `PercentageInteger` for the `brightnessLevel` attribute of the `DimmableLED` class is defined, as shown in Figure 5.7. The `PercentageInteger` data type illustrated in Figure 5.8 contains a boundary definition that may be used to derive test data in later steps (cf. Section 2.7.3, p. 55 ff.).

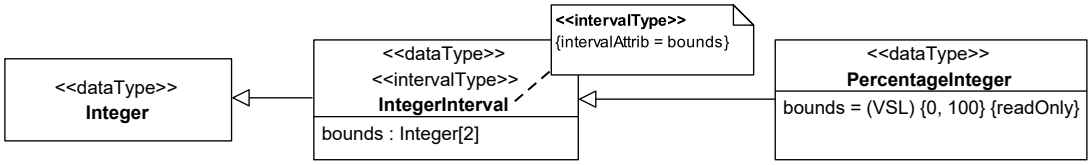


Figure 5.8: Example data type definition using utility types of the MARTE model library (UML 2.5 notation).

5.3.4 Hardware Behavior Package

The purpose of the new *HardwareBehavior* sub-profile is to apply the concept of energy models introduced in Section 5.2.2 by extending the UML metamodel for behavioral state machines [275] with the ability to include descriptions of non-functional aspects for a power consumption estimation. Figure 5.9 shows the stereotypes provided by the *HardwareBehavior* package and the corresponding UML elements to which the stereotypes may be applied.

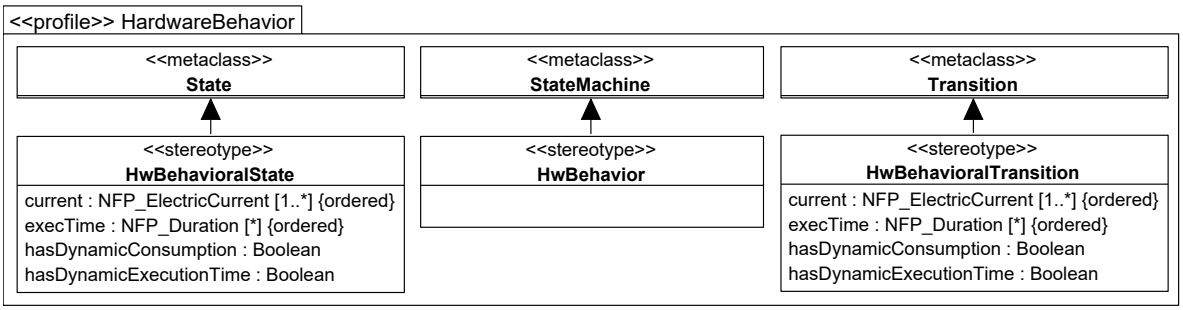


Figure 5.9: *HardwareBehavior* sub-profile package, adapted from [338] (UML 2.5 profile diagram notation). Generalization hierarchies and unused tags have been omitted.

A state machine may be extended with the stereotype `HwBehavior` if it represents an energy model as part of the hardware component model definition. The stereotype contains no additional tags and is mainly used to identify a state machine as an energy model of a hardware component. While serving as a marker for energy models, automatic model-to-model transformations for the analysis process, as mentioned in the developer workflow illustrated by Figure 3.1 (p. 78), can be implemented as part of the proposed power analysis methods introduced in Section 5.4 (p. 135 ff.).

The stereotypes `HwBehavioralState` and `HwBehavioralTransition` provide the same set of tags for behavior modeling and are applied to states and transitions of UML state machines. The most important tags for the proposed power consumption estimation are:

- `hasDynamicConsumption`: This flag indicates if the electric power consumption of the state or transition is static, e.g., fixed value, or dynamic, e.g., described as an expression.

- `hasDynamicExecutionTime`: This tag serves as a flag that indicates if the execution time of the state or transition is variable or static.
- `current`: This tag contains the electric current consumption of a state or transition as a fixed value or an expression.
- `execTime`: This tag contains the execution time of the state or transition. The value of this tag may be specified as a fixed value or an expression. For states, this value may also be empty if the retention time is not specified and depends, for example, on external triggers, e.g., application- or interrupt-triggered.

MARTE distinguishes between static power consumption as a constant, e.g., in standby, and dynamic power consumption as additional power consumed when the component is active and operating (cf. Section 2.1.1, p. 17 ff.). In contrast, the stereotypes `hasDynamicConsumption` and `hasDynamicExecutionTime` of the PAP specify whether the values of the stereotypes `current` or `execTime` are static or depend on the current context of the simulation and, therefore, are changeable within the state or transition. With the concept of modeling dynamic power-related behavior (cf. Section 5.3.5, p. 132 ff.), the PAP can also model dynamic current consumption as an offset of static current consumption similar to MARTE using the tagged value `current`. In this case, `current` contains an expression with at least one reference to a base state defining the static consumption, e.g., an `idle` state of the energy model.

Since the introduced stereotypes `HwBehavioralState` and `HwBehavioralTransition` inherit from MARTE stereotypes such as `ResourceUsage` of the GRM package (cf. Figure 2.10, p. 44), they provide a set of additional tags, e.g., `allocatedMemory`, which are not utilized by the power consumption estimation concept and, thus, have been omitted in Figure 5.9. A complete list of tags for both stereotypes is provided in Appendix B (p. 273 f.).

To ensure full compatibility with MARTE, new tags and data types have been introduced in this thesis to model dynamic and static current consumption. However, MARTE also provides different approaches to model power- and energy-related aspects, which have several disadvantages:

- Due to the generalization hierarchies of the PAP, tags such as `consumption` and `dissipation` from the MARTE `HwResourceService` stereotype are available to model power. However, both tags are based on the data type `NFP_Power`, which, as argued in Section 5.3.2 (p. 126 ff.), is insufficient in this thesis. Dividing the modeling of power into electrical current and voltage provides greater degrees of freedom and also allows more detailed modeling. When referring to Equation (2.1) (p. 18), the value defined for the `consumption` tag defined by the `NFP_Power` data type may be calculated using the tags `supplyVoltage` and `current`. A portion of the power used by a hardware component is lost in the form of heat energy, which can be specified by the tag `dissipation`. However, the measurements carried out in this thesis capture the total current consumption of the system, while no assertions can be made about the portion converted to heat. Therefore, the use of this tag has no advantages.
- The `ResourceUsage` stereotype of the GRM package provides an `energy` tag to specify the amount of energy permanently consumed due to the usage. However, the data type `NFP_Energy` (J) does not fit the presented modeling approach of this thesis. In states, for instance, the retention times are not always known in advance and are determined at runtime. Furthermore, parameters affecting the power consumption of the state may

change during execution. Therefore, it is more suitable to consider the instantaneous power, defined by the electric current consumption (I) and voltage (V), as a parameter detached from the temporal dimension.

Furthermore, the tag `execTime` of the `ResourceUsage` stereotype originally describes the *"time that the resource is in use due to the usage"* [278]. However, while maintaining the intended definition of the `execTime` tag, in PAP, this MARTE tag specifies the execution time of single states and transitions in this thesis, avoiding the definition of a new tag with similar semantics. The following subsection briefly discusses the extension of tag values to include meta data that can be used in the power consumption estimation approach.

Extension of Tag Values with Metadata

NFPs modeled with the PAP may also be extended with additional metadata. For instance, one may specify the source of the value and value ranges, which may be used to derive test cases. In the following, a selected number of possibilities to demonstrate the potential of the profile will be explained while leaving room for further use cases. Examples are explained based on static values, while the general concept can also be applied to expressions.

Referring to the VSL `TupleType` definition (cf. Section 2.6.4, p. 46 ff.), a complete tagged value description, e.g., of the electric current, may be modeled as `current` $= (value = 5, unit = mA, source = meas, statQ = mean)$. The element $source = meas$ describes that the value has been derived from measurements, and $statQ = mean$ indicates that the $value$ is the arithmetic mean. Tags for the *HardwareBehavior* sub-profile are defined with different multiplicities. For instance, the multiplicity range $*$ for the tag `execTime` indicates that it may contain zero or more values. For the `current` tag, the multiplicity range is defined as $1..*$ so that at least one value has to be defined. Due to this, a tag may contain worst-case and best-case values for time- and power-related aspects. For example, a range for an estimated execution time of a transition between 1 and 5 ms may be specified as `execTime` $= ((value = 1, unit = ms, source = est, statQ = min), (value = 5, unit = ms, source = est, statQ = max))$.

Moreover, it is also possible to integrate NFRs of scenarios and energy bug definitions (cf. Section 3.3, p. 84 ff.) into the UML model. Imagine that a power-related NFR with a mean value and an upper limit needs to be modeled. By defining the tagged value `current` as $((value = 5, unit = mA, source = est, statQ = mean), (value = 10, unit = mA, source = est, statQ = max))$, both the expected mean electric current consumption and the maximum current consumption as the upper limit can be modeled, extracted with model-to-model transformations, and used to derive test cases.

When considering scenarios (cf. Section 3.2, p. 81 ff.), tags may also contain multiple static values. Imagine a state of a hardware component model which contains different characteristics depending on the currently active scenario. Tagged values for the electric current consumption with different power consumption levels may be defined as `current` $= ((value = 5, unit = mA), (value = 10, unit = mA))$, and tagged values for the execution time as $execTime = ((value = 30, unit = ms), (value = 15, unit = ms))$. During simulation, an analysis tool would consider the pair $(value = 5, unit = mA)$ and $(value = 30, unit = ms)$ for the first scenario and $(value = 10, unit = mA)$ and $(value = 15, unit = ms)$ for the second scenario, respectively.

Ongoing Example: Behavior and NFP Modeling

The application of the *HardwareBehavior* sub-profile to the ongoing example of the dimmable LED is shown in Figure 5.10. The energy model consists of the two states `On` and `Off`. Instructions defined in the operations `powerOn()` and `powerOff()` (cf. Figure 5.7, p. 128) generate events to initiate state transitions, as explained in Section 5.2.2 (p. 119 ff.). Both states are annotated with the proper stereotype of the PAP. Since transitions are considered instantaneous, the values of the tags `current` and `execTime` are set to 0 mA and 0 ms, respectively. The electric current consumption for the state `Off` is set to 0 mA, while the state `On` has a dynamic consumption indicated by the tagged value `hasDynamicConsumption = true`. In this example, the `current` tag contains an expression instead of a fixed value. The current consumption of this state can vary between 0.05 mA and 5 mA due to the interval definition of $[1, 100]$ (cf. Figure 5.8, p. 129) for the `brightnessLevel` attribute shown in Figure 5.7 (p. 128). The declaration rules and interpretation of such expressions are part of the dynamic behavior modeling covered in the following Section 5.3.5.

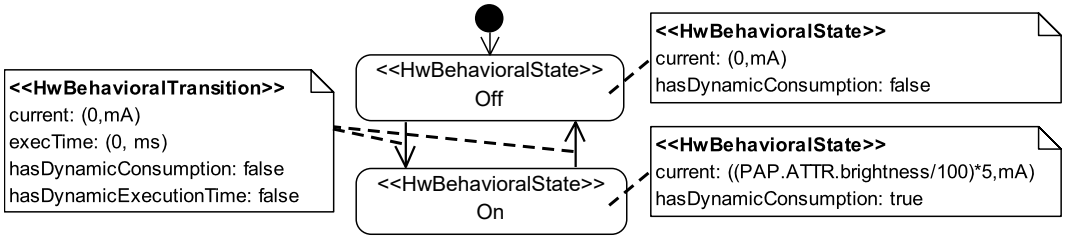


Figure 5.10: Annotated state diagram of the dimmable LED example. Unused tagged values have been omitted to improve legibility (UML 2.5 state machine diagram notation).

5.3.5 Modeling Dynamic Power-related Behavior

For the power consumption estimation process, the electric current consumption and the execution time may be specified with a single numeric value if they are considered static and do not change during execution. However, both aspects may also be affected if, for example, the software model changes configuration parameters during the simulation. In the following, the main concept for modeling the dynamics of hardware components is presented. It is used by the power analysis methods (cf. Section 5.4, p. 135 ff.) within the estimation process.

The tuple notation of NFP data types specified by MARTE [278] and introduced in Section 2.6 (p. 41 ff.) structures the content of tags provided by the `HwBehavioralState` and `HwBehavioralTransition` stereotypes. For the power consumption estimation approach within this thesis, the three elements `value`, `expr`, and `unit` of the tuple notation are required to describe tagged values, for example, $current = (value = 20, unit = mA)$ or $execTime = (expr = 5 \cdot 20, unit = ms)$. While the `unit` element is mandatory, a value either for the `value` or the `expr` element has to be provided depending on the use of static values like fixed numbers or dynamic values defined as expressions. However, the `NFP_CommonType` data type (cf. Section 2.6, p. 41 ff.) provides a set of probability distribution operations that may be used as the `value` of a tag, for instance, `gamma(k : Integer, mean : Real)` for a gamma distribution, `poisson(mean : Real)` for a Poisson distribution with a given mean, or `binomial(prob : Real, trials : Integer)` to specify a binomial distribution with a

probability and number of trials. The MARTE specification does not provide any rules or guidelines for implementing such operations and leaves the interpretation of tagged values, such as $aTag = (value = poisson(1.3), statQ = distrib)$, open to the developers of analysis tools. Additionally, the MARTE specification defines the methodological rule that analysis tools have to compute the *VSL::Expressions::Variables* and return them to the UML model at the start of a VSL evaluation. Due to this, the use of operations as `values` introduces dynamics into UML models. According to [348], the main idea behind the use of probability distribution operations is the definition of a UML generic model as a source, from which multiple UML models as variants with different tagged values may be derived for analysis. However, this is not suitable for considering dynamics of software application models and event-driven approaches at runtime. Instead, the approach in this thesis focuses on expressions that are evaluated online (cf. Section 2.7.5, p. 61 ff.) and re-evaluated for every scenario change and tag value modified during simulation.

In PAP, variables are still based on VSL and use the *Variables* type from the *VSL::Expressions* package [278, 348]. However, their declaration and usage are slightly adapted and differ from the MARTE specification by introducing cross-references between tags of UML elements, such as states and transitions of state machines and attributes of classes, to reflect dynamic behavior in the model definition. Listing 5.1 shows the adapted specification using the *Extended Backus–Naur Form (EBNF)*.

⟨variable-call-expr⟩ ⊨ ⟨variable-name⟩
⟨variable-declaration⟩ ⊨ [⟨variable-direction⟩] `'$'` ⟨variable-name⟩
[`':'` ⟨typename⟩] [`'='` ⟨init-expression⟩]
⟨variable-direction⟩ ⊨ `'in'` | `'out'` | `'inout'`
⟨variable-name⟩ ⊨ [⟨namespace⟩ `'.'`] ⟨body-text⟩
⟨namespace⟩ ⊨ ⟨**pap-prefix**⟩ **[`'.'`** ⟨**pap-postfix**⟩ **]**
`'.'` ⟨**pap-tag**⟩ | ⟨body-text⟩
⟨**pap-prefix**⟩ ⊨ **`'PAP'`**
⟨**pap-postfix**⟩ ⊨ **`'ATTR'`** | **`'SM'`** **`'.'`** ⟨**pap-sm-element**⟩
⟨**pap-sm-element**⟩ ⊨ ⟨body-text⟩
⟨**pap-tag**⟩ ⊨ ⟨body-text⟩
⟨body-text⟩ ⊨ *terminal symbol consisting of a string of characters*

Listing 5.1: Selected parts of the MARTE VSL specification for variables described in EBNF. A definition of the nonterminal symbols ⟨typename⟩ and ⟨init-expression⟩ can be found in [278]. Adaptations in this thesis are highlighted in bold.

By defining a dedicated namespace as a separation between the basic concepts of MARTE and the extension provided by PAP, the compatibility between MARTE and PAP is maintained. The main reason for the extension shown in Listing 5.1 is the definition of links between the object and the meta-level and the modeling of cross-references between elements of the meta-level. Examples of links and cross-references between are provided for the ongoing example at the end of this section. When PAP is used by developers in their MDD workflow as proposed in Section 3.1 (p. 77 ff.), the following set of rules must be respected:

- If a VSL variable for PAP is defined, a specific namespace has to be used.

- The namespace contains at least a `prefix` and a `tag`.
- The `prefix` has to start with a specific term `PAP`.
- The `postfix` can be one of two terms: `ATTR` for attribute or `SM` for state machine. The `postfix` is optional and depends on the type of reference.
- In case the `SM` `postfix` is used, the state or transition name of the state machine (energy model) must be provided.

Ongoing Example: Definition of Variables

Considering the dimmable LED example once again. To access the `bounds` tag of the `PercentageInteger` data type used for the `brigthnessLevel` attribute shown in Figures 5.7 to 5.8, the namespace `Pkg.DimmableLED.brightnessLevel.bounds` has to be used. However, while properties of such data types may be considered constant, values of class attributes become instance-related and differ for each class instance. For example, two instances of the same sensor hardware component model may be configured differently and thus have various current consumption values during runtime. However, such instance-related values are necessary for estimating power consumption, as carried out in this thesis. According to the introduced namespace schema, the reference `PAP.ATTR.brightness` has to be used, for example, to include the value of the attribute `brightnessLevel` in equations. The first part (`PAP`) signals the usage of the corresponding profile, whereas `ATTR` refers to an attribute of the class annotated with the aforementioned stereotypes. The specifier `brightness` refers to the id defined by the `id` field of the `HwAttribute` stereotype applied to an attribute of the UML class (cf. Figure 5.7, p. 128). Note that the interpretation of namespaces is not defined in the MARTE specification and has to be implemented by the analysis tool. During a model transformation, the stereotypes `HwPowerAttribute` and `HwTimingAttribute` may be used to extract the `<variable-direction>`, `<variable-name>`, and `<typename>` described in Listing 5.1 using the tagged values `direction`, `id`, and `type`, respectively. The concept can also be used to specify cross-references between tags of state machine elements. For instance, the variable `PAP.SM.NameOfState.NameOfTag` references the value of `NameOfTag` of the state `NameOfState`. To refer to a tagged value in the scope of the current UML element, the definition `PAP.NameOfTag` may be used.

As shown in Figure 5.10 (p. 132), the electric current consumption for the `On` state can vary between 0.05 mA and 5 mA. Instead of defining multiple states to cover each possible current consumption, PAP prevents state explosion [402] and addresses such dynamic behavior. The expression for the `current` tag is defined as a linear relationship and includes the previously defined reference `PAP.ATTR.brightness`. For non-linear relationships, zero-one indicators may be used, where each variable representing such a zero-one indicator has to be modeled as an attribute of the hardware component model. For instance, the current consumption may be defined as `current` $= (expr = a \cdot 5 + b \cdot 10, unit = mA)$ with a and b as zero-one indicators. While this concept of dynamic behavior introduced by PAP is not specified in MARTE, analysis tools have to keep track of changes for all variables annotated with the `HwPowerAttribute` stereotype during execution to enable a power consumption estimation. Interactions between the software model and a hardware component model may cause the analysis tool to re-evaluate all expressions containing the affected tags.

5.4 Power Analysis Methods

The developer workflow motivated in Section 3.1 (p. 77 ff.) contains simulation and analysis steps for the power consumption estimation of embedded software applications in MDD. As a contribution to address RQ4, the introduced power analysis methods obtain and provide so-called *energy traces* to make the power-related impact of the software application on an embedded system visible. Developers may utilize the energy traces as an indicator for energy bugs (cf. Section 3.3, p. 84 ff.) and to optimize the workflow and the design of the software application model, as illustrated by Figure 3.1 (p. 84). As an umbrella term for the power analysis methods discussed in this section, the term *Software-Model-in-the-Loop (SMiL)* is introduced and describes an in-the-loop testing approach for software application models with integrated energy models. More formally, this term may be defined as follows:

Definition 5.1 *Software-Model-in-the-Loop (SMiL) describes a method in which a* ***software model*** *extended by* ***energy models*** *interacts with a* ***virtual*** *or* ***physical*** *hardware platform based on functional test cases to estimate the power consumption as a non-functional property of the system with a focus on the* ***impact*** *of the software application.*

While energy models have been presented in Section 5.2 (p. 117 ff.), the following sections discuss methods following the SMiL definition. The first method introduced in Section 5.4.1 is denoted as *Indirect Power Analysis (IPA)*. It is used in stages where the embedded system is not yet defined or no physical hardware is available for the evaluation process. With the *Direct Power Analysis (DPA)* introduced in Section 5.4.2, on the other hand, simulated software application models can interact with physical hardware based on real hardware accesses. By this, DPA is able to take hardware-software interactions as a crucial part of the power consumption estimating process [135] into account. Initial versions of the methods have been published in [338, 341] but enhanced in this thesis due to recent research findings.

5.4.1 Indirect Power Analysis (IPA)

The IPA method implements the SMiL approach for virtual hardware platforms (cf. Definition 5.1). For the power consumption estimation, IPA relies primarily on data obtained from energy models during the simulation. Figure 5.11 illustrates the structure of IPA with

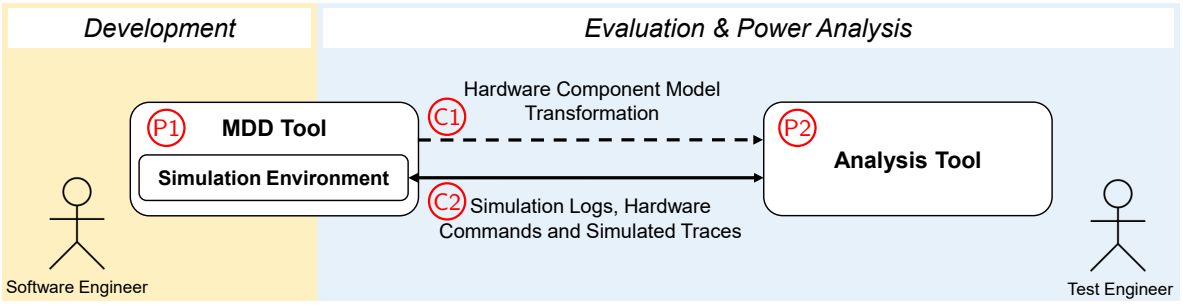


Figure 5.11: *Indirect Power Analysis (IPA)* method to derive energy traces without a connected hardware platform, adapted from [341]. A dashed line describes a data exchange before the execution of a simulation while the solid line indicates a data exchange during the simulation. The red circled elements are used to reference individual parts in this section.

the MDD tool and the analysis tool as the two main parts (P1 and P2) and two communication paths (C1 and C2) between these two parts. During the development phase, the MDD tool annotated with (P1) in Figure 5.11 is used, e.g., by a software engineer to design the system model consisting of the software model and at least one hardware component model. Additionally, the MDD tool provides a simulation environment where the system model can be directly executed. For the evaluation and power analysis, the analysis tool marked as (P2) in Figure 5.11 directly interacts with the simulation environment provided by the MDD tool. As a first step of the analysis process, the analysis tool requires additional information about the SUT. Model-transformation processes of hardware component models (C1) are executed so that the analysis tool gets deep knowledge of the SUT's properties (cf. Section 6.1, p. 140 ff.). A direct communication link (C2) between the MDD tool and the analysis tool is established during the simulation to trace all hardware accesses of the simulated software model, which may lead to changes in the power-related behavior of one or more hardware components. Besides the processing of energy models and simulation logs, simulation data for hardware components can be predefined and passed to the simulation environment if, e.g., a sensor is activated to collect data. By this, scenarios introduced in Section 3.2 (p. 81 ff.) as an environment definition for the SUT may be realized. The communication interface between the simulation environment and the analysis tool is independent of a specific MDD tool.

The analysis tool derives energy traces for the power analysis and the detection of energy bugs automatically or manually, e.g., by a test engineer. Compared to traditional approaches, IPA does not require additional hardware, such as measuring devices. Since all hardware interactions are simulated, the approach may be used for early real-time simulations and design space exploration approaches as long as a hardware component model for each hardware component under consideration exists.

5.4.2 Direct Power Analysis (DPA)

The idea of the rapid prototyping approach (cf. Section 2.9 (p. 72 ff.) is adapted in this thesis to provide a faster approach (compared to HiL) and more realistic approach (compared to MiL) for a power consumption estimation in MDD (cf. Figure 2.21, p. 59). Contrary to the definition provided by [59], the characteristics and constraints of the hardware platform used for the estimation process must be close to those of the intended platform. Otherwise, this type of non-functional test would offer developers no value. Moreover, the concepts of MiL and HiL are utilized in an adapted manner compared to the definitions in Section 2.7.4 (p. 57 ff.) to obtain energy traces as the energy footprint of software application models.

The DPA method implements the SMiL approach for physical hardware platforms by introducing a novel power analysis method. For this, hardware accesses generated by software applications during simulation are forwarded to a hardware platform, denoted as *Model-Testbed*, reproduced, and the results are returned to the simulation environment as traces. While the hardware platform replicates the hardware-software interactions, values for electric current and voltage are measured and continuously recorded, allowing energy traces to be derived. Due to this, DPA may be considered as an extended version or superset of IPA. The basic structure of DPA is illustrated in Figure 5.12.

The parts (P1) and (P2) and the communication paths (C1) and (C2) of DPA shown in Figure 5.12 are similar to the elements introduced by IPA and provide at least the same functionality. However, the analysis tool has been extended to interact with and control the parts (P3) and (P4) in Figure 5.12.

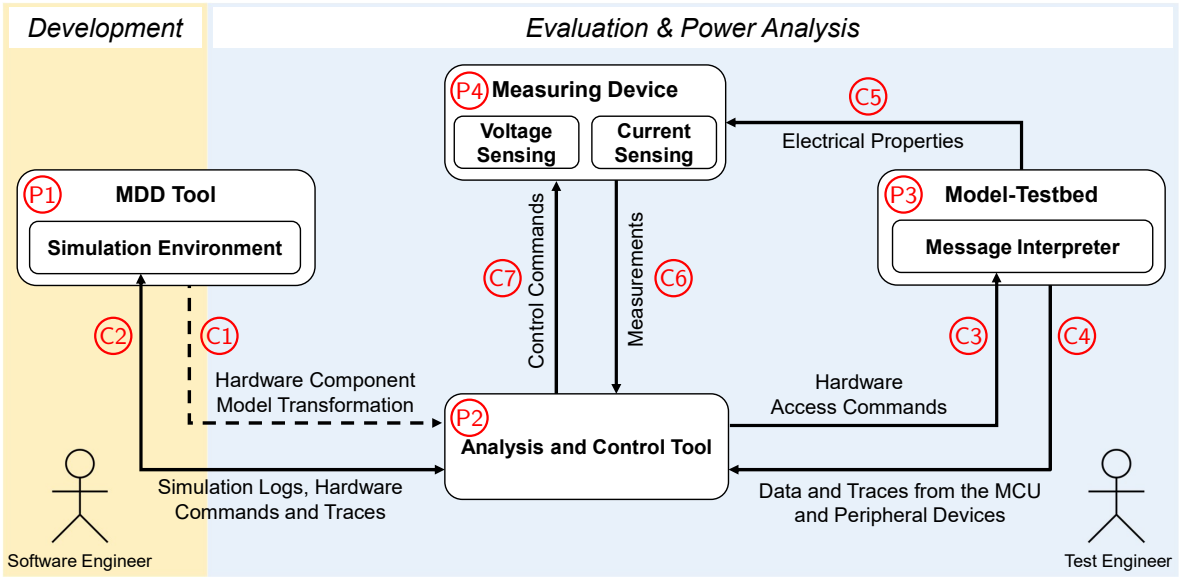


Figure 5.12: *Direct Power Analysis (DPA)* concept to derive energy traces based on interactions with a real hardware platform, adapted from [341]. A dashed line describes a data exchange before the execution of a simulation while solid lines indicate a data exchange during the simulation. The red circled elements are used to reference individual parts in this section.

Besides the processing of hardware component models and the derivation of energy traces, the analysis and control tool (P2) also acts as a bridge between the MDD tool (P1), the *Model-Testbed* (P3), and the measuring device (P4). This provides a more comprehensive performance analysis and results in more detailed energy traces. The proposed approach has been designed as a modular architecture, enabling the rapid development of new setups for specific test cases.

The *Model-Testbed* (P3) executes a *message interpreter* that provides direct hardware access. In DPA, hardware accesses from the software application are directed to the *Model-Testbed* using the communication path (C3), while traces as responses of the *Model-Testbed* are returned over the communication path (C4). The concept of *Model-Testbeds* is not limited to specific controller platforms and allows the dynamic integration of peripheral devices to evaluate their power consumption. The *Model-Testbed* is also suitable for providing actual data (e.g., sensor measurements) to the simulated software model for a more realistic evaluation. With a physical *Model-Testbed* connected to the simulation environment of the MDD tool, the analysis and control tool (P2) is able to compare the power characteristics derived from energy models with real measurements while keeping track of all time-referenced hardware states.

A measuring device (P4) is connected to the *Model-Testbed* for continuous voltage and current measurements (C5). The measured values are accumulated (C6) and evaluated by the analysis and control tool. Since the tool is aware of the internal state of each hardware component model instance based on the trace data sent by the simulation environment, states and transitions of each hardware component model instance can be mapped to a specific time or time frame of the measurement and combined with the measured values. Additionally, the analysis and control tool uses control commands to initiate measurements and configure the measuring device in general, e.g., ranges and resolution of the obtained data (C7).

Since DPA is focused on estimating power consumption, it is sufficient to replicate hardware-software interactions instead of executing the entire software application on the *Model-Testbed*. Each instance of a hardware component model is coupled with a specific peripheral device of the *Model-Testbed*. Without the need to generate, adapt, compile, and flash the software application model before each test, the DPA defines a rapid prototyping approach and enables a faster and more realistic evaluation of the complete system compared to traditional approaches (cf. Section 2.8, p. 65 ff). The use of a physical hardware platform also increases the probability of detecting (hardware-related) energy bugs (cf. Section 3.3, p. 84 ff.).

Additionally, runtime monitoring is suitable for the DPA method. However, contrary to the definition discussed in Section 2.7.5 (p. 61 ff.), the monitor component in this thesis is not generated and implemented by the analysis tool (P2). Instead, the analysis tool may be parameterized and configured based on scenarios, NFRs, and hardware component models, which corresponds to the concept of an interpreted monitor [117].

Test cases performed with DPA may also be based on scenarios but with a limited scope compared to IPA. To fully support the concept of scenarios, an additional environmental model is required that also provides the ability to influence the physical test environment of the *Model-Testbed* directly. However, this results in a more complex setup and requires the extension of the concept shown in Figure 5.12. While currently not part of DPA, the extension will be a subject of future work. The DPA method is part of the proof-of-concept implementation discussed in depth in Chapter 6 (p. 139 ff.).

Chapter 6

Prototype Implementation

This chapter describes the prototype implementation of the *Direct Power Analysis (DPA)* method for the power consumption estimation in MDD as a part of the evaluation defined by the developer workflow (cf. Figure 3.1, p. 78). While the software and hardware modeling process is covered by the case study in Chapter 7 (p. 175 ff.), this section focuses on implementing main components, tools and protocols to address RQ4 as shown in Figure 6.1. The red circled numbers (1–8) in Figure 6.1 reference individual parts of the DPA method introduced in the following sections.

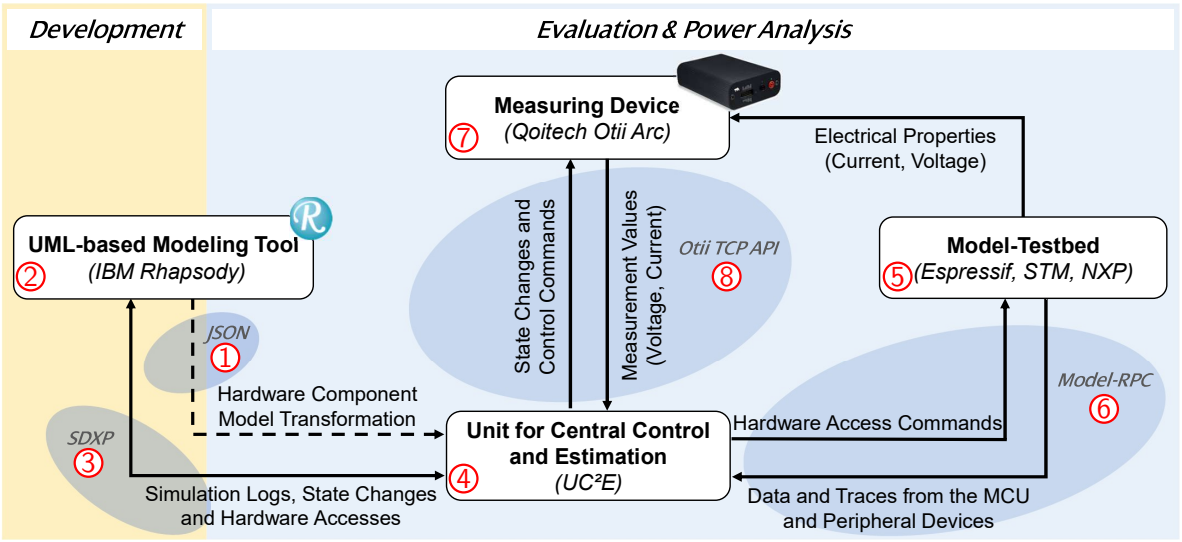


Figure 6.1: DPA concept using IBM Rhapsody as MDD tool and Qoitech Otii Arc as measuring device. Solid arrows indicate connections used during the simulation and evaluation, while the dashed arrow describes a manual step performed before the evaluation. The red circled numbers (1–8) reference individual parts.

Section 6.1 discusses the textual representation of hardware component models for the model transformation process, referred to as (1) in Figure 6.1. Additionally, the developed model transformation plug-in enhancing the MDD tool IBM Rhapsody (2) to provide a hardware component model exchange with the UC^{2}E tool (4) is briefly discussed. Section 6.2 introduces the *Simulation Data eXchange Protocol (SDXP)* (3) and the messaging framework

to provide tracing and access to the *Model-Testbed*. The concept of a policy-oriented HAL to exchange logs and simulation data for hardware-software interactions is covered by Section 6.3. Section 6.4 briefly introduces the main functions of the UC^{2}E tool (4) as a reference implementation of the analysis and control tool. Finally, Section 6.5 covers the developed *Model-Testbed* (5) from a software and hardware perspective. Additionally, a *Remote Procedure Call (RPC)*-based protocol (6) to achieve communication with *Model-Testbeds*, is provided. Measurements are obtained by a Qoitech Otii Arc measuring device [313] (7) and can be accessed through the associated Otii *Transmission Control Protocol (TCP)* server [312] (8) as part of the Otii desktop application. Parts of the implementations and protocols of this section have been published in [339, 340, 341, 391].

6.1 Model Transformation

The power analysis methods introduced in Section 5.4 (p. 135 ff.) rely on data exchange before and during the simulation to perform power consumption estimations. For the execution of model transformations, access to the logical model structure is necessary, which has to be provided by APIs of MDD tools. The transformation of hardware component models is implemented as a two-step process containing a model-to-text and a text-to-model transformation based on the textual representation format covered in Section 6.1.1. For model-to-text transformation, Section 6.1.2 presents an extension of the IBM Rhapsody MDD tool that is provided as a plug-in. The textual representation is the primary format for the model exchange denoted as (1), while the enhancement of IBM Rhapsody is part of (2) in Figure 6.1.

6.1.1 Textual Representation

This section introduces the structure of the textual representation used in the model transformation process. For the exchange of hardware component models between IBM Rhapsody [164] and the UC^{2}E tool, an exogenous model-to-model transformation has been defined to map a UML model to a C++-based textual model of state machines (m/UML $\rightarrow$ m/C++). Considering the developer workflow shown in Figure 3.1 (p. 78), a model-to-text transformation from UML into the JSON-based textual representation ($m(s)/UML \rightarrow m(s)/JSON$) is performed by IBM Rhapsody as the first part of the two-step process. This is followed by a text-to-model transformation from the JSON representation to C++ ($m(s)/JSON \rightarrow m(s)$/C++), the language in which the UC^{2}E tool is developed. By this, the UC^{2}E tool remains MDD tool independent. The text-to-model transformation is covered in Section 6.4 (p. 155 ff.).

With *XML Metadata Interchange (XMI)*, a standardized model exchange format specified by the OMG exists, which is natively supported by popular MDD tools, including IBM Rhapsody (cf. Table A.2 in Appendix A.2, p. 265 f.). Current XMI parser and generator implementations of MDD tools are designed to encode and decode complete UML models. Considering hardware component models, this includes the entire UML class and the UML state machine model. However, for the power consumption estimation, the UC^{2}E tool only requires a subset of the hardware component model definition, e.g., the structure of the energy model, power-related tagged values, and attributes annotated with the PAP. Approaches such as [413] enhance the abilities of XMI parsers to process UML models partially, which would reduce the required memory for the parsing process and lower the execution time. However, those approaches have not yet been integrated into MDD tools for UML mentioned in Appendix A.2 (p. 265 ff.). Due to this, JSON has been selected instead of XMI as the data

interchange format for the prototype implementation in this thesis. Compared to *Extensible Markup Language (XML)* as the data format for XMI documents, JSON is considered more lightweight, uses fewer resources, and can be encoded and decoded faster [266].

Behavior-related characteristics are the most important aspects of a hardware component model for the power consumption estimation approach. Since the energy model mainly covers behavioral and non-functional aspects, the basic structure of the JSON-based interchange format for the model transformation shown in Listing 6.1 is derived from the state machine representation. The top-level element is specified as an array of JSON objects representing hardware component models. A single hardware component model consists of a `name` which is the name of the UML class and objects for attributes, settings, states, and transitions. In the following, the objects of the JSON structure shown in Listing 6.1 are described in detail. A coherent and complete example generated by the developed plug-in for the dimmable LED introduced in Section 5.3 (p. 124 ff.) is provided in Appendix C.1 (p. 275 f.).

```
{
   "HardwareComponentModels": [{
      "Name": "NameOfBaseClass",
      "Attributes": {
         "_comment": "cf. Listing 6.2"
      },
      "Settings": {
         "_comment": "cf. Listing 6.3"
      },
      "States": {
         "_comment": "cf. Listing 6.4"
      },
      "Transitions": {
         "_comment": "cf. Listing 6.5"
      }
   }]
}
```

Listing 6.1: Structure of the JSON-based interchange format for hardware component models.

Attributes and Settings

All attributes with the annotated stereotypes `HwPowerAttribute` and `HwTimingAttribute` are extracted as JSON objects and attached to the main `Attributes` object. Each attribute is identified by a unique `AttributeName` name corresponding to the name used within the UML class definition. The inner structure of a single `Attributes` object shown in Listing 6.2 and consists of the following properties:

- `id`: The attribute's unique id extracted from the value of the id tag provided by the `HwAttribute` stereotype. As stated in Section 5.3.3 (p. 127 ff.), the `id` is used as a reference to the value of the attribute within a class instance and may be included in expressions of other tags of the PAP.
- `dataType`: A property to specifiy the data type, typically an `integer` or `real`.
- `value` (optional): A property containing the initial value for the attribute.

```
1  {
2     "HardwareComponentModels": [{
3        "Attributes": {
4           "AttributeName": {
5              "id": "id",
6              "dataType": "aType",
7              "value": 0
8           }
9        }
10    }]
11 }
```

Listing 6.2: Basic structure of the `Attributes` object.

The `Settings` object maps the basic power-related characteristics of a hardware component model as key-value pairs. The basic idea is to group all properties into a settings object, which apply to the entire hardware component model regardless of the particular state or transition. Typically, tags of the `HwAbstraction` stereotype applied to UML classes are part of the `Settings` objects. A small example is shown in Listing 6.3. The value of the key-value pair in line 4 is a composite data type consisting of an `integer` or `real` value and a unit as a `string` data type (cf. Section 2.6.4, p. 46 ff.).

```
1     {
2        "HardwareComponentModels": [{
3           "Settings": {
4              "id": "(value, unit)"
5           },
6        }]
7     }
```

Listing 6.3: Basic structure of the `Settings` object.

States and Transitions

To trace and analyze the dynamic behavior of the SUT for the power consumption estimation during simulation, information about states and transitions annotated with the `HwBehavioralState` and `HwBehavioralTransition` stereotypes are included in the model transformation process and added to `States` and `Transitions` objects. The basic structure of a `States` object is shown in Listing 6.4. As a design decision, instead of an unnamed array of states, the `stateId` has been introduced as a property to ease access to individual elements during parsing. Each state consists of the properties `name`, `id`, and `behavior` and is identified by the `stateId` (line 4). The `id` must be unique, can be freely assigned, and is primarily used in `Transitions` objects to specify their source and destination. In IBM Rhapsody, for instance, a *Universally Unique Identifier (UUID)* is generated for each model element of the Rhapsody model, which has been used for the `id` fields in the example shown in Appendix C.1 (p. 275 f.). The `behavior` property describes the power-related behavior and is also an object consisting of the

```
{
   "HardwareComponentModels": [{
      "States": {
         "stateId": {
            "name": "stateName",
            "id": "uniqueId",
            "behavior": {
               "current": "(value, unit)",
               "execTime": "(value, unit)",
               "hasDynamicConsumption": false,
               "hasDynamicExecutionTime": true
            }
         }
      }
   }]
}
```

Listing 6.4: Basic structure of the `States` object.

`current`, `execTime`, `hasDynamicConsumption`, and `hasDynamicExecutionTime` properties as direct mappings of the corresponding tagged values specified by the `HwBehavioralState` stereotype.

As shown in Listing 6.5, the structure of a `Transitions` object also includes a `name` and a `behavior` property, as well as the following additional properties:

- `initialTransition`: True if the current transition is a initial transition, false otherwise.
- `fromState`: The unique id of the source state. Unused for initial transitions.
- `toState`: The unique id of the destination state.

```
{
   "HardwareComponentModels": [{
      "Transitions": {
         "transitionId": {
            "name": "transitionName",
            "initialTransition": true,
            "fromState": "",
            "toState": "297bd30c-c077-4e57-9b16-4906cab429c9",
            "behavior":{
               "current": "(value, unit)",
               "execTime": "(value, unit)",
               "hasDynamicConsumption": true,
               "hasDynamicExecutionTime": false
            }
         }
      }
   }]
}
```

Listing 6.5: Basic structure of the `Transitions` object.

6.1.2 Enhancement of the MDD Tool

For the model-to-text transformation, a plug-in for IBM Rhapsody [164] has been developed. The plug-in uses the provided Java API and is integrated into the MDD tool as a *Helper* [166]. In general, *Helpers* represent custom programs attached to IBM Rhapsody to extend its functionality. The integration of *Helpers* is realized via so-called *helper files* where, for example, the name, the Java main class, and menu commands of the *Helper*, are specified as shown in the middle part of Figure 6.2. Since the model transformation plug-in is integrated into the context menu of the GUI, developers can start the model transformation of hardware component models right out of the MDD tool. In IBM Rhapsody, profiles such as the PAP (cf. Section 5.3, p. 124 ff.) have MDD tool-specific properties in addition to the basic properties provided by the UML specification [275], e.g., to include the local path of helper files in profile specifications. As a result, the model transformation plug-in is loaded into IBM Rhapsody automatically whenever the PAP is added to an MDD project. This process is illustrated in Figure 6.2. The Java source code files on the left side of Figure 6.2 represent the developed plug-in. The helper file shown in the center of Figure 6.2 (`transformation.hep`) contains two entries to specify the main class and the file path of the plug-in. Within the properties window of the PAP profile in IBM Rhapsody on the right side of Figure 6.2, the helper file may be integrated using the path to the helper file.

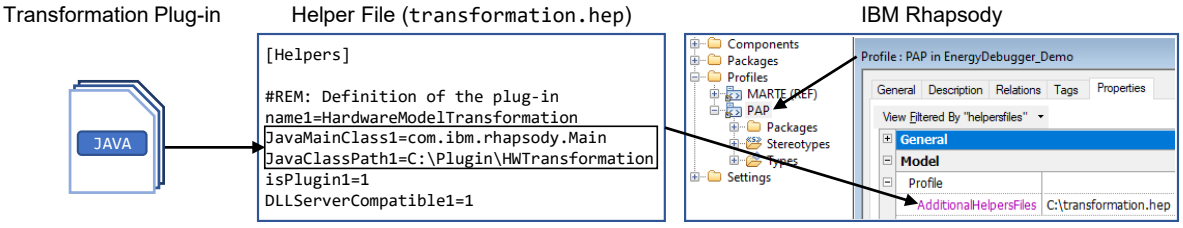


Figure 6.2: Integration of Helpers into IBM Rhapsody

To to access and parse the user-developed Rhapsody model, the developed plug-in uses the IBM Rhapsody Java API. The Rhapsody model is an IBM-specific interpretation of UML models hidden behind their graphical representations. More information about the IBM Rhapsody Java API and best practices can be found in [89, 167]. When the transformation plug-in is executed, the developer may choose a destination path and a file name for the model-to-text transformation. Afterward, the plug-in iterates through the currently active MDD project while using stereotypes of the PAP as indicators for annotated classes and state machines to collect the required information of hardware component models. The process described by the UML activity diagram in Figure 6.3 is executed for each UML class in the UML model annotated with a `HwAbstraction`-based stereotype (cf. Figure 5.6, p. 127). In the first step, fundamental data, such as the class name, are acquired and stored in a list. In the next step, all attributes annotated with the `HwPowerAttribute` or `HwTimingAttribute` stereotypes are analyzed, and the content of the `id`, `type`, and `value` tags are saved along with their attribute names. Afterward, the algorithm checks if a state machine with annotated `HwBehavior` stereotype exists. If a corresponding state machine is found, states and transitions are analyzed. Based on the stored information, the JSON-based file is generated in the last step of the algorithm. If no state machine exists, the plug-in terminates without producing an output.

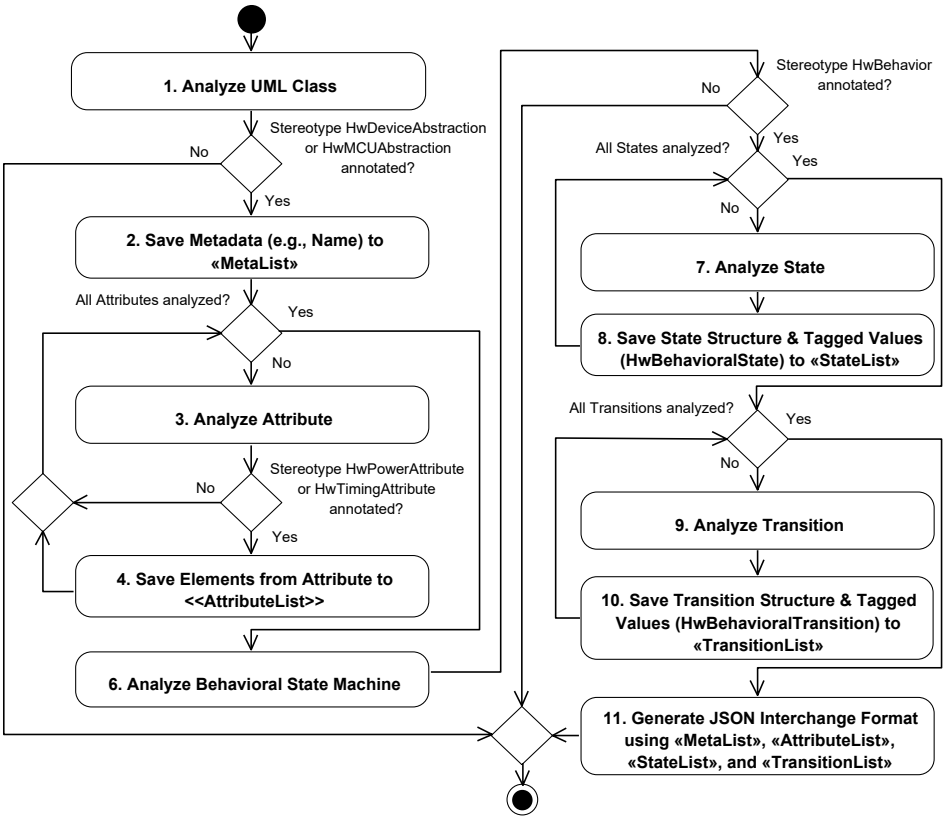


Figure 6.3: Sequence of the JSON-based interchange file creation process (UML 2.5 activity diagram notation), published in [340].

6.2 Data Exchange

To implement the power analysis methods introduced in Section 5.4 (p. 135 ff.), a set of communication protocols has been developed to enable interactions between the simulation environment of the MDD tool, the UC^{2}E tool, and the *Model-Testbed*. Section 6.2.1 introduces the *Simulation Data eXchange Protocol (SDXP)* as a data exchange format between the simulation environment and the UC^{2}E tool. The SDXP is used for the communication denoted as (3) within the DPA method illustrated in Figure 6.1, while the messaging framework is part of the system model developed in (2). A reference implementation for integrating SDXP into a UML-based project is presented in Section 6.2.2.

6.2.1 Simulation Data eXchange Protocol (SDXP)

This section covers the *Simulation Data eXchange Protocol (SDXP)* specification. In order to achieve bidirectional communication between the simulation environment and external analysis tools, such as the UC^{2}E, the SDXP specifies three message types for the data transmission: *Register*, *Behavior*, and *Action*. These message types enable a well-defined exchange of trace information, simulation data, e.g., measurements, and control commands for real-time interaction with a *Model-Testbed*. The specification is not limited to a particular representation format such as JSON, XML, or *Comma-separated Values (CSV)* and is compatible with any

two-way communication interface. For the proof-of-concept implementation in this thesis (cf. Section 7, p. 175 ff.), the simulation environment and UC^{2}E tool are connected via a TCP socket using a CSV-based representation format. In the following, the message types of SDXP are explained in detail.

Register Message Type

The *Register* message type is used whenever a hardware component model instance is created within the simulation. For the registration of a hardware component model instance at the UC^{2}E tool, the name of the instance and the actual hardware component model type must be provided. Table 6.1 shows the specification of the *Register* message type, which contains a `MessageId` field, mainly used to distinguish between the *Register* (0), *Behavior* (1), and *Action* (2) message types, and a timestamp of the registration event. The field `Device` contains the name of the hardware component model instance, while the `Type` field specifies the actual base name of the hardware component model.

Field	Datatype	Description
MessageId	Integer	Message identifier, always set to 0 for *Register* message type.
Timestamp	Integer	Timestamp of the message (simulation time).
Device	String	Name of the instance.
Type	String	Name of the hardware component model base type.
Binding	String	(Optional) hardware binding information.

Table 6.1: Basic structure of the *Register* message type.

Since the approach is designed to support and distinguish multiple instances of the same basic hardware component model, this runtime information must be provided for analysis and tracing. For instance, if an embedded system consists of two sensors of the same type, two register messages with different values for the `Device` and `Binding` fields are generated and sent to the UC^{2}E tool before the software application model may interact with those instances. By this, the UC^{2}E tool is able to distinguish between instances of a component at runtime without requiring additional configuration effort by the developer for each test case execution. The `Binding` field is optional and contains information about the physical wiring and hardware interfaces as key-value pairs. When using JSON object literals, a binding for a hardware component model can be specified as, for example, `{"interface":"pin","address":{"gpio":26}}` to describe that the hardware component is connected to the GPIO with the number `26`. The interface `pin` states that the GPIO is configured as a simple switch that can be turned on and off, e.g., to control an `LED`. The UC^{2}E tool (cf. Section 6.4, p. 155 ff.) may use the binding to configure the *Model-Testbed* automatically using the *Model-RPC* protocol specified in Section 6.5.4 (p. 168 ff.).

Behavior Message Type

The *Behavior* message type is primarily used to report state and attribute changes of hardware component model instances so that a tracing and external analysis outside the simulation environment may be achieved. Table 6.2 contains the associated fields for the *Behavior* message type. Besides the fields `MessageId`, `Timestamp`, and `Device`, which are identical to those of the*Register* message type (cf. table 6.1), three additional fields are introduced to

the *Behavior* message structure, namely `State`, `Transition`, and `Attributes`. For accurate tracking, a *Behavior* message must contain a timestamp (simulation time) at which the event occurred and the name of the affected hardware component model instance. The `State` field contains the `id` of the new power state. If the hardware component model instance executes a non-instantaneous transition between two power states, the `Transition` field is used instead.

Field	Datatype	Description
MessageId	Integer	Message identifier, always set to 1 for *Behavior* message type.
Timestamp	Integer	Timestamp of the message (simulation time).
Device	String	Name of the instance.
State	String	(Optional) id of the power state.
Transition	String	(Optional) id of the executed transition.
Attributes	String	(Optional) Key-value pairs of altered attributes.

Table 6.2: Basic structure of the *Behavior* message type.

The `Attributes` field of a *Behavior* message contains the new values of altered attributes required to calculate power consumption or execution time for states or transitions of a hardware component model (cf. Section 5.3, p. 124 ff.). By this, the UC^{2}E tool is notified during the simulation and can recalculate the affected parameter values of the energy model in near real-time. When a state change results in altered parameters, a single Behavior message is generated containing the new state and all altered attributes, e.g., by using the two fields `State` and `Attributes`.

Action Message Type

An *Action* message is generated whenever a direct interaction between the hardware component model and the physical counterpart of the *Model-Testbed* is required. For this, a bidirectional transmission is implemented based on a request-response pattern. Table 6.3 shows the fields of the *Action* request message.

Field	Datatype	Description
MessageId	Integer	Message identifier, always set to 2 for *Action* message type.
ActionId	Integer	Action identifier, specifies sub-type for MCU behavior (0), peripheral device behavior (1), request data (2) and send data (3).
Device	String	Name of the instance.
RemoteAction	String	Operation to be called.
Parameters	String	Key-value pairs of parameters for the operation.
ResponseId	Integer	(Optional) Set, if a response from the *Model-Testbed* is required.

Table 6.3: Basic structure of the *Action* request message type.

This message type distinguishes between messages targeting the MCU or a peripheral device. Compared to peripheral devices, the MCU of the *Model-Testbed* is a special case and is treated separately by the UC^{2}E tool, which is further explained in Section 6.4 (p. 155 ff.). In total, four sub-types have been derived and are specified by the value of the field `ActionId`. The first two types, with the id 0 (MCU) and 1 (peripheral device), summarize all messages that directly affect the state of a hardware component, e.g., turning a peripheral device on or off.

The remaining sub-types specify interactions with a peripheral device to request (2) or send (3) information. For instance, an *Action* message enables interaction with the *Model-Testbed* to, e.g., trigger a measurement, request sensor values, send data via a communication interface, or write configurations. The field `RemoteAction` defines the actual operation executed in the *Model-Testbed* environment. If parameterized operations are used, e.g., the value to be written, the `Parameters` field is set with key-value pairs. For obtaining data via a response from the *Model-Testbed*, the `ResponseId` field is set.

Field	Datatype	Description
MessageId	Integer	Message identifier, always equal to 2 for *Action* message type.
ResponseId	Integer	Same as the response id from the request.
Result	String	Result of the action (e.g., measurement value, bytes sent, ...).

Table 6.4: Basic structure of the *Action* response message type.

Since hardware component models can generate messages simultaneously, the response format shown in Table 6.4 also includes a `ResponseId` field with a value equal to the corresponding value of the request message. While *Behavior* messages are generated within state machines due to behavior changes, *Action* messages, on the other hand, may also be generated within operation calls of the hardware component model instance, e.g., if attributes annotated with the `HwPowerAttribute` or `HwTimingAttribute` stereotype are altered. Since not every message results in a state change and state changes do not necessarily require communication with the *Model-Testbed*, e.g., in case of time-triggered transitions (cf. Section 5.2.2, p. 119 ff.), actions and behavioral changes are considered separately by SDXP. Hence, the execution of a single operation may result in multiple *Behavior* and *Action* messages.

6.2.2 Messaging Framework

A simple SDXP-based messaging framework has been developed to allow UML models to communicate with external applications during the simulation in MDD tools, such as IBM Rhapsody. Designed as an extension, it may reduce the the need for manual modification of the original UML model as much as possible. Since the framework is written in C++, it can be integrated into the IBM Rhapsody project as an external source or by reverse engineering. For the data exchange, the framework connects to the UC^{2}E tool (cf. Section 6.4, p. 155 ff.) via a socket connection at the beginning of a simulation. It also provides functions to generate SDXP messages. Figure 6.4 illustrates the UML class diagram of the messaging framework with the two classes `ModelLogger` and `ModelConnector`. In UML and IBM Rhapsody, external source artifacts are annotated with the «`File`» stereotype and included in the UML model by defining dependencies with annotated «`usage`» stereotype. While the `ModelLogger` class in Figure 6.4 provides operations to send and receive data following the SDXP message types, the `ModelConnector` class manages the socket connection to the UC^{2}E tool.

Method calls of the messaging framework must be integrated into the opaque behavior of UML elements such as operations, states, or transitions. For instance, both classes are used within the proof-of-concept implementation of the policy-oriented HAL introduced in the following Section 6.3 (p. 149 ff.). The UML model of the case study presented in Section 7.2 (p. 176 ff.) has been manually extended with required method calls to enable simulation data exchange. However, specific keywords or additional stereotypes may be defined in future work

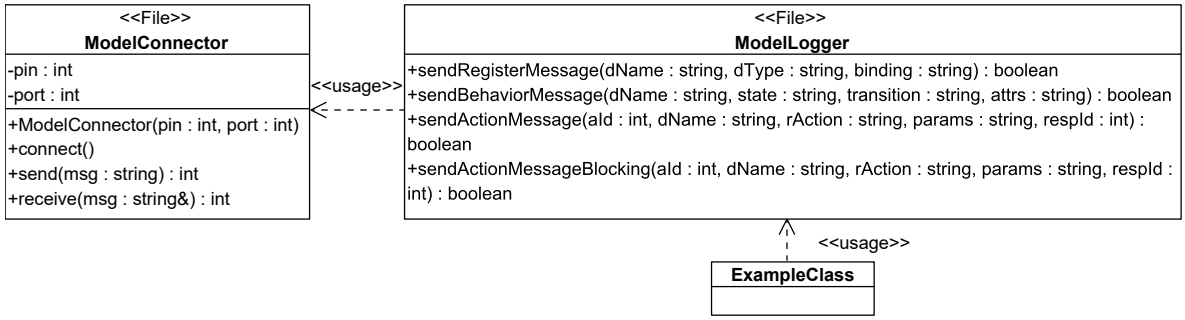


Figure 6.4: UML classes of the messaging framework (UML 2.5 class diagram notation).

to automatically generate SDXP-related method invocations within opaque behavior based on *Helpers* and model transformations before the actual simulation is executed. A concept based on such model-to-model transformations has been presented in [162].

6.3 Policy-oriented Hardware Abstraction Layer

This section introduces the policy-oriented HAL as a new approach for the abstraction of hardware accesses which may be integrated into models developed by (2) in Figure 6.1. The term *policy-oriented* originates from the concept of policy-based design [11] as a variant of the strategy design pattern [131], in which strategies are specified at compile time. Instead of strategies, policies in the context of the HAL refer to communication interfaces. The initial concept of the HAL has been published in [391] but is extended to be integrated into the overall approach of this thesis. The policy-oriented HAL focuses on communication interfaces, such as GPIO, UART, or I^2C, which are used by peripheral devices while also considering energy-related aspects as a contribution to answer RQ3 and RQ4. In addition to the theoretical concepts presented in this section and in [391], a C++ reference implementation of the policy-oriented HAL is presented. The reference implementation defines a simulation-specific variant using the data exchange framework and SDXP (cf. Section 6.2.2, p. 148 ff.) as the first step to enable communication between hardware component models and their counterparts on the *Model-Testbed*.

A general overview of HAL implementations is provided in Section 6.3.1, followed by an in-depth description of the concepts and structure of the policy-oriented HAL presented in Section 6.3.2. In Section 6.3.3, the UML profile as a model representation of the policy-oriented HAL is introduced. Finally, an basic application example of the policy-oriented HAL in provided in Section 6.3.4.

6.3.1 Overview

As a widely used programming interface for UNIX-like operating systems, the *Portable Operating System Interface (POSIX)* I/O API [170] has been defined, allowing stream and file-like access to individual devices by providing the five operations `open`, `close`, `read`, `write`, and `ioctl`. The *Common Microcontroller Software Interface Standard (CMSIS)* [20], a programming interface driven by ARM for their processor platforms, provides functions for controlling the power consumption of the corresponding internal hardware unit (on-chip) in a simplified

manner. However, all mentioned approaches are focused on the MCU instead of the complete system, including connected and external peripheral devices.

In contrast to traditional HAL implementations such as POSIX I/O and CMSIS, the presented HAL provides a multi-tier architecture. It relies on policy-based access patterns for communication interfaces and connected peripheral devices. It also considers existing power-related functions of peripheral devices, which are often unused in traditional approaches or limited to on and off states or other unspecified low-power states. Moreover, the policy-based HAL is vendor-independent and may be used for an abstraction of internal and external *Class 1* to 3 hardware devices (cf. Section 5.2.1, p. 117 ff.) which can be controlled w.r.t their power consumption as demonstrated in [391].

6.3.2 Three-layered Architecture

The access to external peripheral devices from the software application layer can be seen as a vertical flow that proceeds from a high-level through various low-level software layers until the hardware is finally addressed. To control such devices, a lower hardware communication layer must be accessed. The presented HAL is designed as a hierarchical approach and follows the *Dependency Inversion Principle (DIP)* as one of the five SOLID principles[1] [238] to realize a policy-based access pattern. By inverting the dependencies of the underlying hardware layer, the flexibility and portability are increased while the effort to modify the hardware platform is minimized. Although DIP has been known for a long time, to the best of the author's knowledge, it has not been consistently used in device driver design.

By introducing an additional abstraction to describe the interface required to access the external peripheral device, the policy-oriented HAL is able to consider peripheral devices that support different hardware interfaces. For instance, a sensor may be connected to an I^2C or SPI interface. Although the physical wiring and communication protocol are different, the functionality of the HAL is independent of the hardware interface, and the details are hidden from the application point of view. This may minimize the effort for developers to modify any application logic when the hardware configuration changes. The three-layered architecture of the policy-oriented HAL outlined in Figure 6.5 provides functional access and power-related control for peripheral devices. The *device layer* represents the highest and most abstract layer of the architecture and consists of the following elements:

- `ConnectedDevice`: An abstract class as the base representation of a specific peripheral device. A `ConnectedDevice` requires at least one access policy for the interaction with physical hardware and may use an additional interface to enable and disable the peripheral device.

- `Device`: This class represents a concrete specification of a `ConnectedDevice`. This class has to provide an interface to the power management functions of the peripheral and may offer additional control functions.

- `Access`: A representation of an access policy, typically defined as an abstract class providing a set of virtual operations to access the peripheral device.

[1] SOLID is a mnemonic acronym in software engineering for five design principles to enhance the understandability of object-oriented design and consists of the single-responsibility, open–closed, Liskov substitution, interface segregation, and dependency inversion principles [238].

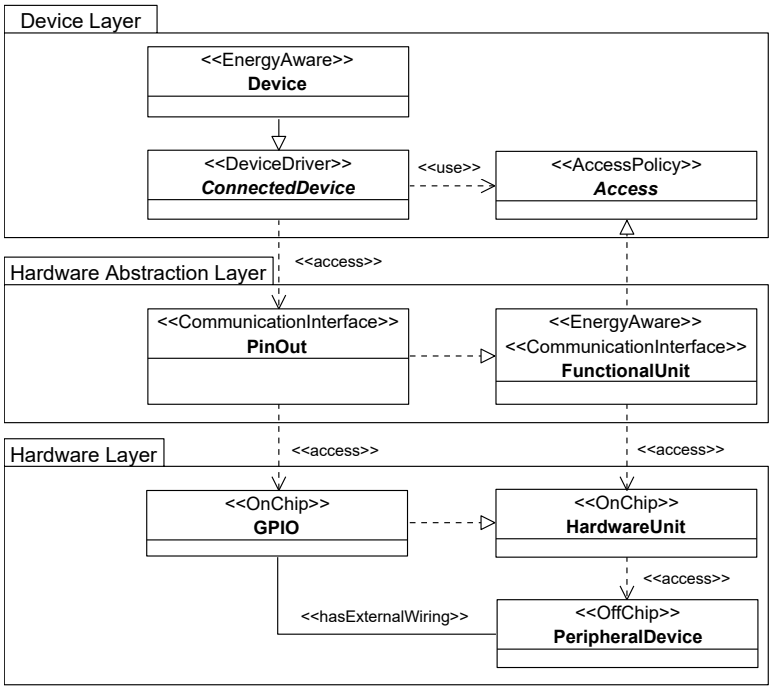


Figure 6.5: Three-layered architecture of the policy-oriented HAL (adapted from [391]; UML 2.5 package diagram notation).

In general, an access policy defines the requirements of a `ConnectedDevice`, as a proxy of a hardware component at the software layer, for a communication interface in an abstract manner. The proposed policy-oriented HAL provides the following access policies:

- `BitAccess`: Policy for the basic interface to set and clear a single pin. This type is typically used to manage the behavior of GPIOs.
- `ValueAccess`: Policy to read or write single values, e.g., ADC or DAC.
- `RegisterAccess`: Policy to access the value of a single addressable register. Bus systems like I^2C and SPI typically realize this type of access policy.
- `CharacterAccess`: Policy that supports read and write operations with ordered sequences of unstructured characters (bytes). Typically, serial interfaces such as UART and sockets implement this policy type.

The layer in the center of Figure 6.5 outlines the core HAL, which provides a low-level abstraction of communication interfaces such as I^2C or UART, implements functional access by realizing a specific access policy to control the hardware unit and is responsible for the power consumption control. Additionally, classes in this layer follows the DIP. A specific communication interface is represented as `FunctionalUnit`. This class typically contains platform-specific commands for interacting with the physical hardware module (`HardwareUnit`). The class `PinOut` implements a `FunctionalUnit` and contains the actual configuration (e.g., GPIO number or UART bus identifier). The lowest layer represents the physical hardware and consists of the following elements:

- `HardwareUnit` (on-chip): Integrated unit to enable communication between hardware components. The unit can be powered on and off, e.g., by using clock gating. Examples of such units are GPIO, UART, and I^2C.
- `GPIO` (on-chip): Describes a physical pin provided by the MCU.
- `PeripheralDevice` (off-chip): A device physically connected to the MCU by a `HardwareUnit`, such as sensors and actuators. The device may have internal power management functionalities (cf. Section 5.2.1, p. 117 ff.).

In summary, the use of the policy-oriented HAL offers several advantages. By inverting the dependencies, the device layer is independent of the concrete realization of lower-level communication interfaces and the underlying hardware platform. Due to this, the functionality of a `Device` may be detached from the concrete communication interface and its configuration, which may significantly improve the portability, interchangeability, modifiability, and reusability of software applications. Suppose the concept of the policy-oriented HAL is adapted for other hardware platforms. In that case, only the implementation of the `FunctionalUnit` has to be adjusted, e.g., by defining the use of platform-specific operations and data types. The stereotypes used in Figure 6.5 are part of the developed UML profile of the policy-oriented HAL and are explained in the following Section 6.3.3.

6.3.3 Model Representation

A UML profile illustrated in Figure 6.6 has been designed to define a model representation of the policy-oriented HAL on the meta-level for UML-based models in MDD. By providing a set of stereotypes for the elements of the policy-oriented HAL, UML classes representing parts of the HAL may be annotated to enhance the modeling process for developers and to enable additional model transformation steps if required.

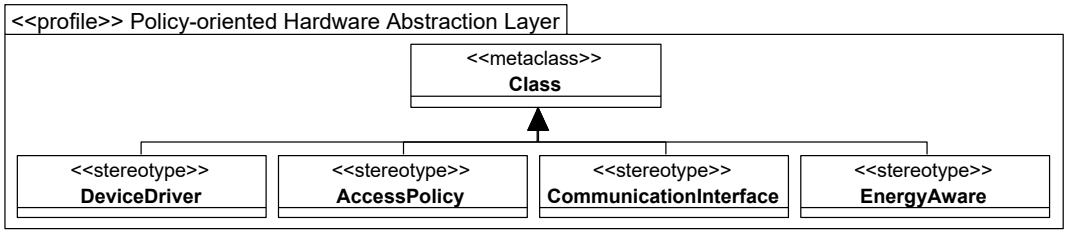


Figure 6.6: Overview of the policy-oriented HAL profile (adapted from [391]; UML 2.5 profile diagram notation).

UML classes annotated with the «`DeviceDriver`» stereotype correspond to an entity representing the functional interface of a peripheral device, which is usually off-chip. In the context of this thesis, hardware component models are typically annotated with the «`DeviceDriver`» stereotype. Moreover, communication interfaces required by hardware component models may be expressed by access policies. In an UML model, policies are annotated with the «`AccessPolicy`» stereotype. Classes annotated with the «`CommunicationInterface`» stereotype represent communication interfaces, also described as functional units of the system, realizing one or multiple access policies. A class stereotyped as «`EnergyAware`» has the ability to control the dynamic power consumption of a connected entity. This stereotype

does not define an expected behavior but assumes that energy-aware peripheral devices have at least two distinct energy states, e.g., `on` and `off`. Note that the concept behind the stereotype «`EnergyAware`» refers to energy models and the PAP. Due to this, hardware component models are always considered energy-aware.

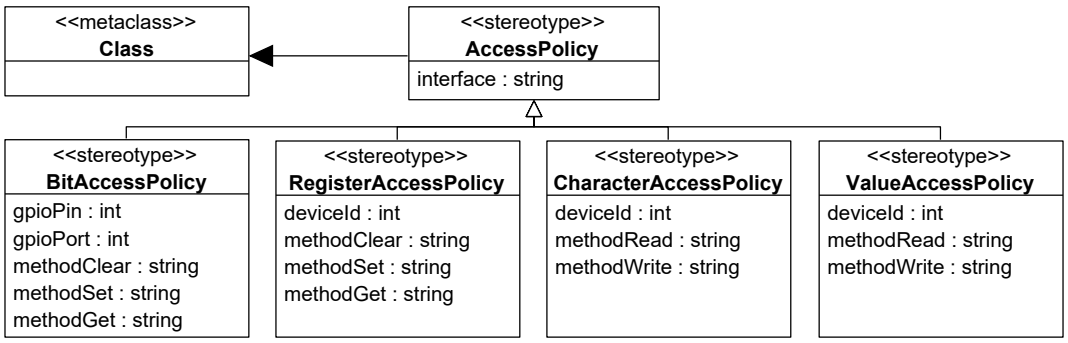


Figure 6.7: Stereotypes defining `AccessPolicy` variants (UML 2.5 class diagram notation).

For each access policy introduced in Section 6.3.2 (p. 150 ff.), a distinct stereotype has been defined, as shown in Figure 6.7. The tags of the stereotypes contain information about the interface type, e.g., GPIO or UART, the physical connection, e.g., the GPIO pin or the hardware-specific identifier, e.g., `UART0`, and the provided methods of the class required for the interface-specific communication, e.g., `read`, `write`, and `set`. Such information may be extracted and used to generate platform-specific configurations and operation calls as opaque behavior similar to concepts presented in [163].

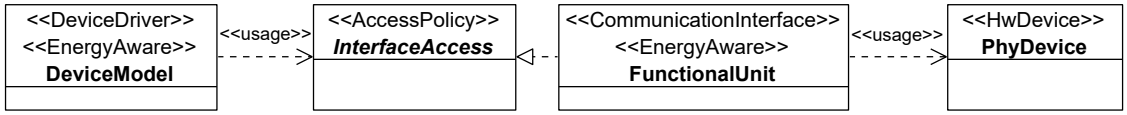


Figure 6.8: Exemplary implementation of the policy-based device pattern (adapted from [391]; UML 2.5 class diagram notation).

Figure 6.8 shows the exemplary usage of the stereotypes and the relationships between the elements, expressed as UML classes. The `DeviceModel` in Figure 6.8 represents a software element, such as a hardware component model or a simple device driver, that encapsulates access to a peripheral device via specific interfaces for which specific requirements exist. Such a requirement may be expressed by an access policy that specifies communication details for the interaction with a peripheral device. As communication interface, a `FunctionalUnit` realizes an access policy and provides access to a physical hardware device (`PhyDevice`). The `PhyDevice` is extended with the «`HwDevice`» stereotype of the MARTE profile [278] to indicate a resource attached to the hardware platform.

6.3.4 Application Example

An example implementation of the policy-oriented HAL is shown in Figure 6.9 for a `Lamp` class as a proxy for an LED, externally connected via a GPIO (`PinOut`). The GPIO hardware unit implements the `BitAccess` policy by using a `pinWrite` method to control. In object-oriented

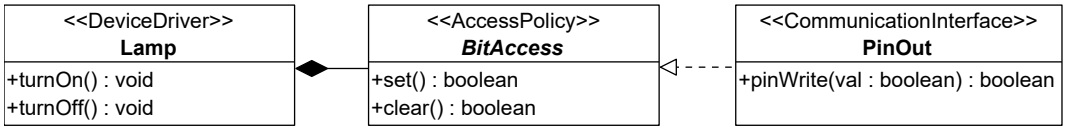


Figure 6.9: Policy-oriented HAL example for a lamp class with `BitAccess` policy, adapted from [391] (UML 2.5 class diagram notation).

programming, a policy can be realized by an interface class. From a source code point of view, policy-based design is usually associated with the C++ programming language [11]. For instance, in this thesis, a policy may be described by a template with concepts to define a named set of requirements expressed as template parameters avoiding the disadvantages of dynamic polymorphism compared to a class declaration with pure virtual methods [403]. Listing 6.6 shows the declaration of the `BitAccess` policy.

```
1  /* GPIO Access. */
2  template<typename T>
3  concept BitAccessable = requires(T t) {
4     { t.set()} -> std::same_as<bool>;
5     { t.clear()} -> std::same_as<bool>;
6  };
```

Listing 6.6: Pin-like access policy example as a `BitAccess` policy variant, published in [392].

A type `T` realizes the `BitAccessable` concept if it implements the methods `set()` and `clear()`. Both methods must return a boolean value. On the device layer, the device class `Lamp` in Figure 6.9 may use this specific implementation of the `BitAccess` strategy to turn a lamp on and off, as shown in Listing 6.7.

```
1  template<BitAccessable PIN>
2  class Lamp {
3     public:
4     explicit Lamp(std::unique_ptr<PIN> p):_pin(std::move(p)) {}
5     void turnOn() { _pin->set(); }
6     void turnOff() { _pin->clear(); }
7     protected:
8
9     /* Pin to set/clear. */
10    std::unique_ptr<PIN> _pin;
11 };
```

Listing 6.7: Implementation of a device layer class in C++ using a predefined `BitAccess` policy, adapted from [392].

A more complex example is shown in Figure 6.10. A `Device` class as a proxy of the Bosch BME280 sensor [53] is connected to the MCU using an I^2C hardware unit, as one of two supported `RegisterAccess` policies. Since the communication interface implements

a master-slave architecture, an additional device connector class (`I2CDeviceConnector`) is added. It acts as a link between the communication interface at the HAL core and the device driver class.

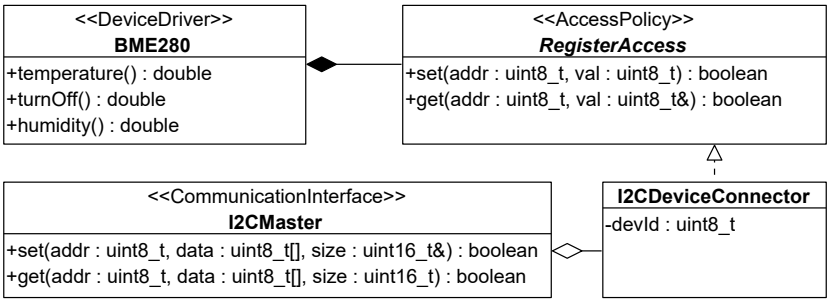


Figure 6.10: Policy-oriented HAL example for a sensor connected via I²C (adapted from[391]; UML 2.5 class diagram notation).

6.4 Unit for Central Control and Estimation (UC²E)

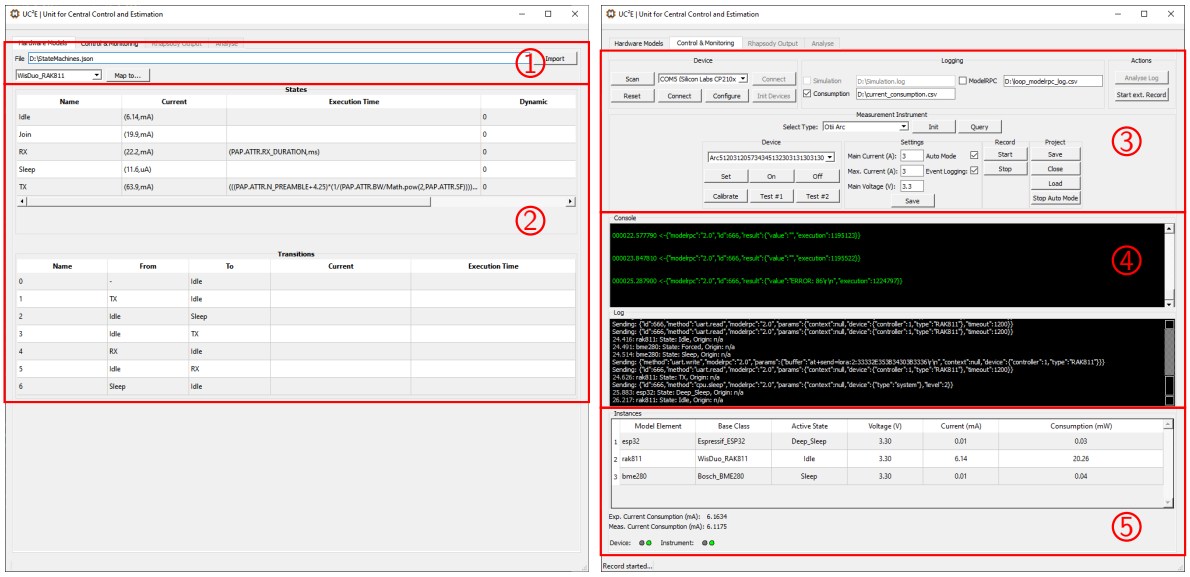
The UC²E tool is the central component for the power analysis methods presented in Section 5.4 (p. 135 ff.) and allows developers to estimate the power consumption of software application models. For this, the UC²E tool is able to consider and apply scenarios and to perform an online or offline analysis. The range of functions also includes:

- The detection of energy bugs.
- The control of the communication between the system model simulated by MDD tools and the *Model-Testbed*.
- The management of connected measuring devices.

In the following sections, key features and functions of the UC²E tool are highlighted. Section 6.4.1 presents an excerpt of the GUIs as an interface for developers to configure and execute test cases, estimate power consumption, and detect energy bugs. Details of the communication between the UC²E tool and other components of the power analysis methods are explained in Section 6.4.2, while Section 6.4.3 describes the integration of measuring devices. Finally, Section 6.4.4 covers selected details of the power estimation process.

6.4.1 Graphical User Interface

The GUI defines the primary interface for developers to prepare the test environment, connect the *Model-Testbed*, and configure the measuring unit. Figure 6.11 shows two exemplary views as an excerpt of the developed UC²E tool. With the first view shown in Figure 6.11(a), developers are able to perform a text-to-model transformation of hardware component models (1) based on the JSON-based model interchange format introduced in Section 6.1.1 (p. 140). As described in Section 3.1 (p. 77 ff.), the transformation process is necessary to initially configure the UC²E tool with hardware component models of the SUT to be used within the simulation and evaluation. The lower part of Figure 6.11(a), highlighted as (2), displays the result of the



(a) (b)

Figure 6.11: Screenshots showing the hardware component model import view and the configuration and estimation view of the UC²E tool, adapted from [341]. The red circled numbers (1–5) are used to discuss parts and functions of the tool.

model transformation process. By selecting a hardware component model using a drop-down menu at the left side of (1), the states and transitions are displayed in (2) as a table, along with their power-related characteristics. If necessary, the developer can adapt the characteristics of single states and transitions by editing the corresponding cells in (2). Furthermore, the wiring (e.g., interface, port, address) for each hardware component model may be defined manually using the button on the left part of (1) as a direct mapping between a hardware component model and the corresponding counterpart as part of the *Model-Testbed*. As an automatic process, wiring information may also be provided during the simulation based on the SDXP defined in Section 6.2.1 (p. 145 ff.).

Figure 6.11(b) shows the application's main view, in which the developer can prepare the overall system for the power analysis and perform an online evaluation. In part (3) of Figure 6.11(b), a connection to the *Model-Testbed* can be established and logging options selected. Additionally, in the lower section of part (3), a set of buttons is provided to configure and control the Qoitech Otii Arc measuring device [313]. Part (4) shows the live logging with two output windows for the *Model-Testbed* communication and the interaction with the simulation environment, including request and response messages in both formats, *Model-RPC* and SDXP. Part (5) of Figure 6.11(b) contains a table showing all hardware component model instances, their active states, and expected power consumption in real-time. Additionally, the estimated and measured current power consumption values of the SUT are displayed to provide a real-time estimation. In additional views not explicitly shown in Figure 6.11, the developer can access a runtime monitoring component as a live graph showing the overall power consumption, a log analyzer, and a time-based graphical comparison between the estimated and measured power consumption.

6.4.2 Communication Principles

In DPA, the UC²E tool has to interact with three different systems: the MDD tool, the *Model-Testbed* and a measuring device. Since the connection between the UC²E and the MDD tool (cf. (3) in Figure 6.1, p. 139) is based on a TCP socket, the simulation and the analyzes can be executed on different systems. A specific version of the policy-oriented HAL (cf. Section 6.3, p. 149 ff.) has been developed to connect the simulated system model with the UC²E tool. To abstract hardware accesses, the HAL uses SDXP messages (cf. Section 6.2.1, p. 145 ff.). Following a low-level approach, only basic methods of communication interfaces such as UART or I²C (e.g., `read()` and `write()`) have to be considered instead of high-level and hardware component-specific operation calls (e.g., `measure()`).

The HAL is specifically designed to be used in simulations of MDD models and can be replaced with a platform-specific variant in later MDD phases. Since no rework of the software model is required when the HAL is exchanged, the software model remains platform-independent. A serial connection based on UART is used to establish a connection between the *Model-Testbed* and the UC²E tool. Other communications links, such as sockets, can easily replace the serial connection if the UC²E tool and the *Model-Testbed* are executed and connected to different hosts. In this case, additional latencies, e.g., due to the transmission medium, must be considered, which may negatively affect the power estimation process. The

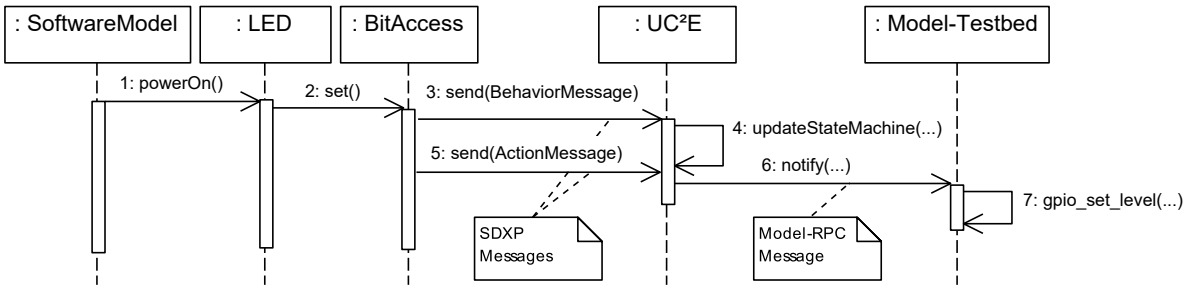


Figure 6.12: Sequence diagram to enable a peripheral device. (UML 2.5 sequence diagram notation)

UML sequence diagram in Figure 6.12 shows the interaction between the system model, the UC²E tool, and the *Model-Testbed* to enable a peripheral device connected via a GPIO. The system model consists of the software application model, a LED hardware component model with access to a GPIO provided by a `BitAccess` instance as part of the policy-oriented HAL.

A UML sequence diagram for a request-response example is shown in Figure 6.13. This scenario describes the interactions between the system model, the UC²E tool, and the *Model-Testbed* to trigger a measurement of a sensor. High-level method calls may result in multiple sequential read and write commands. In the case of the method `measure()` as the first step of the UML sequence diagram shown in Figure 6.13, several I²C `set` and `get` commands may be executed if, for instance, multiple registers need to be written, read, or monitored to confirm successful operations. For readability, the communication between the sensor instance and the physical sensor in Figure 6.13 is shown with a single `set` or `get` operation. The execution of the `measure()` function generates a single SDXP *Behavior* message and each `set` or `get` operation of the HAL implementation initiates an *Action* message. All SDXP messages are processed by the UC²E tool and transformed into *Model-RPC* messages if necessary.

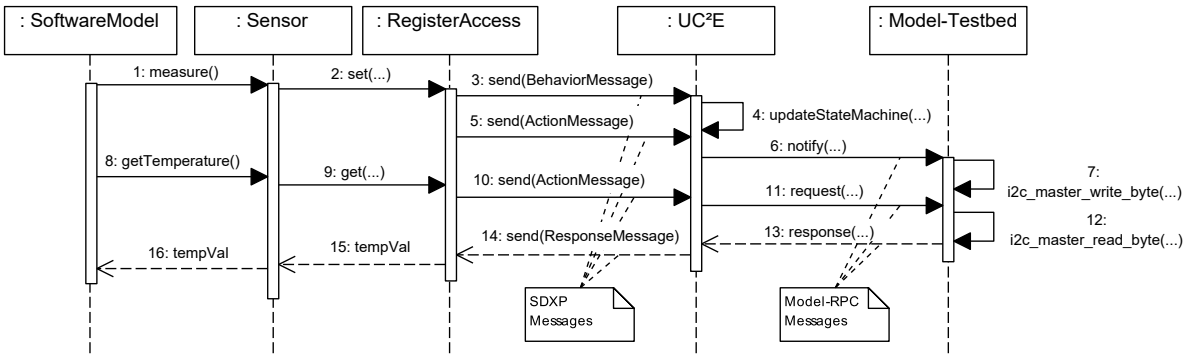


Figure 6.13: Simplified sequence diagram to retrieve data from the *Model-Testbed*. (UML 2.5 sequence diagram notation)

6.4.3 Integration of Measuring Devices

The UC^{2}E tool is not limited to a specific type of measuring device. By implementing universal communication protocols like the standard commands for programmable instruments [176] and the virtual instrument software architecture standard [75, 177], measuring devices from different manufacturers can be supported. However, the Qoitech Otii Arc [313] does not support the aforementioned communication protocols. Instead, Qoitech provides a dedicated software to manage and control the Otii Arc which also includes the TCP-based server [312]. The server is marked as (8) in Figure 6.1 (p. 139) and provides an interface for third-party programs like the UC^{2}E tool (marked as (4) in Figure 6.1, p. 139) to communicate with the measuring device and retrieve measurement data.

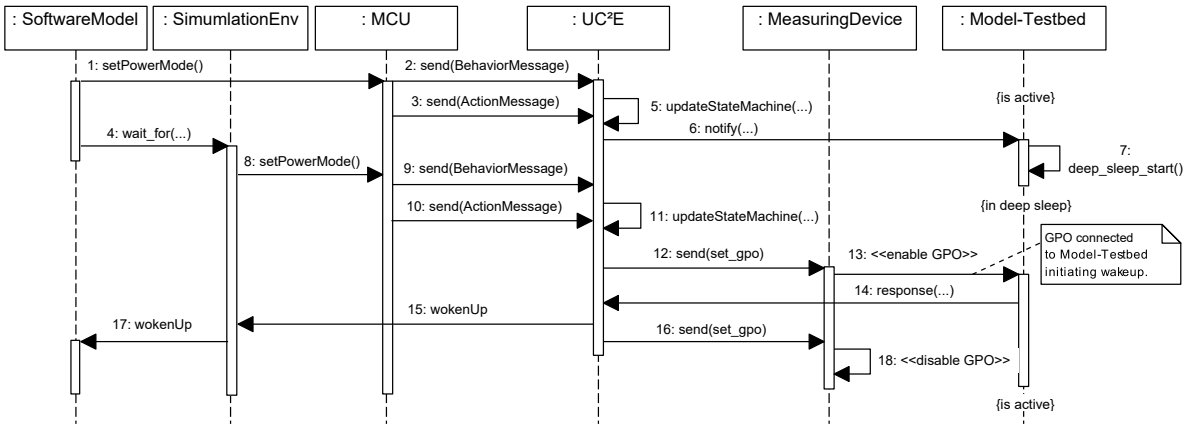


Figure 6.14: Sequence diagram to demonstrate power mode changes of the *Model-Testbed* (UML 2.5 sequence diagram notation).

The life cycle of the *Model-Testbed* is directly affected by the operating state of the MCU. Different power modes of the MCU must be supported to achieve a realistic power estimation when measuring the *Model-Testbed* during simulation. An additional concept has been developed to achieve a defined behavior of the *Model-Testbed* while the MCU operates in low-power modes. The *Model-Testbed* is able to switch from low-power modes to active

mode even if the MCU is not operating and not executing any software. In order to reproduce changing operating states of the simulation, especially the change from low power to an active operational mode, the *Model-Testbed* provides external triggers. The Otii Arc also offers extended functionalities via a so-called expansion port, including two *General Purpose Outputs (GPOs)* that can be accessed using the provided server. By this, the UC²E tool is able to control the complete behavior of the *Model-Testbed* at runtime. Figure 6.14 shows a UML sequence diagram where the software application initiates the command to set the MCU into sleep mode. To simplify the simulation at this point, the software application waits for the configured time before the MCU is re-enabled. A blocking function call is used for this action so that the execution of the software application model will not continue until the MCU on the *Model-Testbed* enters the active state.

6.4.4 Power Consumption Estimation

The power estimation capacities of the UC²E tool differ depending on whether IPA or DPA is used. However, both methods use hardware component models to obtain information about the MCU and peripheral devices as part of the simulated system model. For a power consumption estimation by the UC²E tool, several subjects have to be considered, which are thematically grouped and explained in the following. The first two subjects deal with the mapping of hardware models and the consideration of expressions as dynamic energy- and time-related behavior of states and transitions. Afterward, the process of the UC²E tool to perform the power consumption estimation within IPA and DPA is explained in detail.

Mapping of Hardware Component Models

As a result of the text-to-model transformation of hardware component models based on the model transformation format (cf. Section 6.1.1, p. 140 ff.), the UC²E tool creates a map of hardware components for which instances can be initiated during the simulation. The structure of the C++-based model as part of the UC²E tool is illustrated as a UML class diagram in Figure 6.15. Each hardware component model is represented as a `HwModel` whose structure with classes like `HwState` and `HwTransition` is similar to state machines. Name-value pairs and arrays of attributes are mapped to `HwAttribute` and `HwSetting` classes and integrated as properties of the `HwModel`. A complete element-wise mapping between the JSON and C++ representations is provided in Table C.1 of Appendix C.2 (p. 277 f.).

Consideration of Expressions

As described in Section 5.3 (p. 124 ff.), expressions based on VSL are used to describe dynamic behavior within states and transitions. The VSL already supports basic mathematical operations, and a subset of more complex functions is provided in the specification [278]. However, the specification does not attempt to provide a complete set of complex functions nor limits the extensions of the VSL to support domain-specific functions. For instance, functions such as `exp10()`, `exp2()`, and `expE()` may be created to extend the arithmetic operation of exponentiation for different basis (10, 2, e). As a design choice for the prototype implementation, instead of defining new functions for every mathematical operation required in expressions, a JavaScript-based definition is used to simplify parsing and interpretation by the UC²E tool. For instance, the function `Math.pow(b,e)` is used in tagged values of the PAP to specify exponentiation with the basis `b` and exponent `e`.

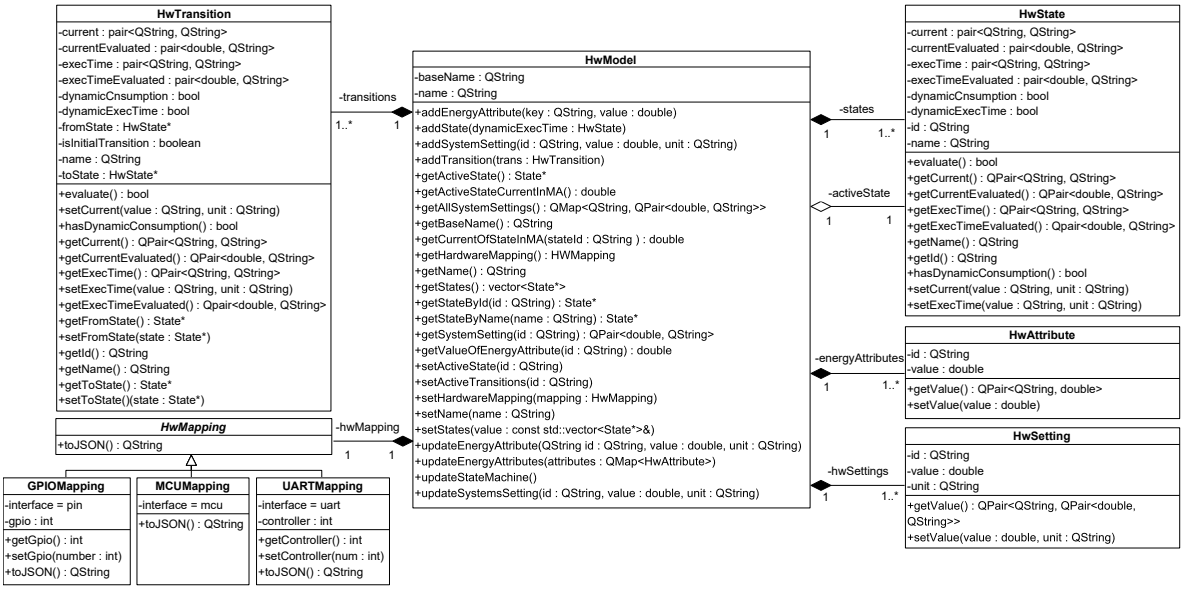


Figure 6.15: Mapping of hardware component models in C++ (UML 2.5 class diagram).

Estimation Process of IPA and DPA

For IPA, the power estimation process is based solely on behavioral changes in hardware component models and their modeled consumption values. The UC^{2}E tool is able to draw line charts for an online estimation and to generate logs for offline analysis. Additional properties must be considered for hardware components with their own complex logic to achieve realistic interactions in IPA. However, this increases the need for more complex hardware models.

Imagine a sensor connected via an I^{2}C interface. To achieve a low-level communication between the virtual hardware component and the hardware component model, the UC^{2}E tool has to provide a virtual memory map containing all registers which are externally reachable via the I^{2}C interface. By this, the system model might be able to retrieve data such as measurement values during the simulation. Such a virtual model has been successfully implemented for the Bosch BME280 [53] and integrated into the UC^{2}E tool for the evaluation of the power consumption estimation approach in Chapter 7 (p. 175).

Since no hardware platform is used and hardware accesses are virtual, only software-related energy bug candidates may be detected. When keeping track of all hardware component models during simulation, an active component during a low-power phase when executing a test case might indicate an energy bug. The developer must analyze the behavior to evaluate whether it is a an error or intended behavior. For instance, a sensor may be left enabled, e.g., if the expected sleep period is too short and, thus, it is too expensive to turn it off and on.

In DPA, the UC^{2}E tool uses two different sources for power estimation. To obtain measurement data during the test case execution, the UC^{2}E tool queries the Otii server within an interval of 1 ms. Beside the table-based live view of hardware components shown in Figure 6.11(b) (p. 156 ff.), the measured data are combined with the modeled values of hardware component models into a single line graph (online) and log file (offline) to provide developers with energy traces for the power consumption estimation.

Based on energy traces, the UC^{2}E tool is also able to automatically detect candidates for energy bugs. An abrupt increase of the measured current consumption without a prior state change and message exchange may indicate a hardware-related energy bug (Type *A* or *B*). Moreover, inconsistencies between the expected and measured values may indicate additional hardware-related energy bugs. As another example, the software application model sets the MCU into a low-power mode. As a result, the UC^{2}E tool expects a lower current consumption of the overall system. If the measured value does not indicate a power consumption drop to the expected level, a software-related energy bug (Type *C* or *D*) might exist in the design of the software model. In both cases, further investigations of the developer are necessary to exclude an inaccuracy in the energy model or a simulation error. Since the UC^{2}E tool traces active states and power consumption levels for all hardware component models during the simulation, it may be able to identify the affected hardware component causing the misbehavior and provide initial feedback to the developer. For complex use cases with multiple hardware components involved, future work may extend this approach by implementing a matching algorithm since the deviation detected by the UC^{2}E tool is roughly equal to the single or the sum of several `current` tagged values of states or transitions.

In general, the UC^{2}E tool supports scenario definitions as introduced in Section 3.2 (p. 81 ff.), specifically to detect energy bugs. However, some of the requirements and constraints specified in scenarios require extended environmental control, which cannot always be guaranteed if DPA is used in field experiments instead of laboratory tests. IPA allows a direct influence of hardware models if this is required to fulfill a specific scenario. For instance, a virtual hardware model of a sensor may return readings derived by the UC^{2}E tool from the currently active scenario. As a message bridge, the UC^{2}E tool might also be extended for fault injections.

6.5 Hardware-based Model-Testbed

This section covers the development of the hardware-based *Model-Testbed*. Following the SMiL approach (cf. Definition 5.1, p. 135), the concept of a hardware-based *Model-Testbed* allows a direct interaction with the SUT, e.g., simulated software model, function-wise and energy-wise. As a contribution to overcome RQ4, *Model-Testbeds* are used in conjunction with DPA to evaluate software application models in early MDD phases. In Figure 6.1, the *Model-Testbed* is located at (5).

A brief overview of the *Model-Testbed* concept is given in Section 6.5.1. Section 6.5.2 focuses on the software layer and introduces the architecture and functions of the message interpreter, denoted as *firmware* in the following sections. For the hardware layer discussed in Section 6.5.3, the *Model-Testbed* concept has been applied to three MCU architectures from different vendors, namely Espressif Systems ESP32 [111], NXP LPC54114 [269], and STMicroelectronics STM32L476 [371]. As a universal and lightweight communication protocol to interact with *Model-Testbeds* (cf. (6) in Figure 6.1, p. 139), *Model-RPC* is introduced in Section 6.5.4.

Initial concepts of *Model-Testbeds* and *Model-RPC* have been published in [339, 340, 341, 393] while a supervised master's thesis [62] provides contributions by extending the firmware and realizing the advanced *Model-Testbed* concept focusing on rapid prototyping and developer productivity.

6.5.1 Overview

The main goals of the *Model-Testbed* concept are to enable dynamic and near real-time hardware-software communication and realistic interaction with the environment. The definition of *Model-Testbeds* follows a minimalistic, modular and interchangeable design concept. The term minimalistic describes the property of a *Model-Testbed* to use a minimum and only the necessary amount of hardware components. Especially in the layouts of commercial evaluation boards, the system is equipped with additional components such as converters, regulators, interfaces, debuggers, and sometimes even sensors and actuators, which are not necessary for the intended evaluation and can negatively influence the measurement and estimation of power consumption. For MCUs as the core of the *Model-Testbed*, for instance, the reference implementation provided by datasheets only consists of some additional resistors, switches, and capacitors. The terms modular and interchangeable describe concepts of a *Model-Testbed* where components, such as individual peripheral devices and the MCU, can be reconfigured or replaced to perform specific tests, e.g., for design space exploration. To meet the requirements of rapid prototyping discussed in Section 2.9 (p. 72 ff.) as idea to overcome RQ4, adding and replacing individual components should be possible without much integration effort. Following the concept of a breakout board, the MCU needs to provide as many GPIOs and access to functional blocks as possible to connect and integrate external peripheral devices. In addition, the developed firmware should ensure that each *Model-Testbed* can be controlled similarly, regardless of the underlying hardware layer. The general setup of the *Model-Testbed* is sketched in Figure 6.16 using the SysML notation [279]. If no power consumption estimations and evaluation of NFRs have to be performed, the *Model-Testbed* can communicate directly with the SUT, e.g., a simulated UML model of the system. However, for the power consumption estimation based on DPA, the UC^{2}E tool, as an additional component, is required to act as a message bridge for the two components.

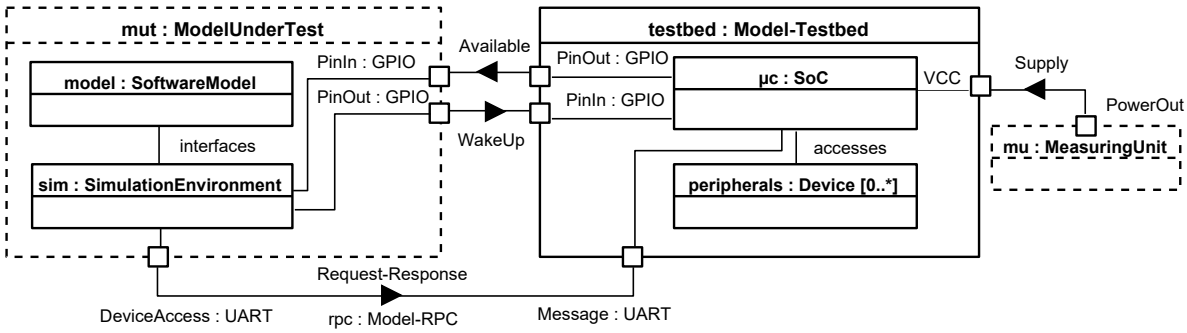


Figure 6.16: Block diagram showing principal parts and usage of the *Model-Testbed*, adapted from [341] (SysML 1.6 internal block diagram notation).

In Figure 6.16, UART is used as a communication protocol, which can easily be replaced by other concepts such as sockets. This illustrates the universal purpose of the *Model-Testbed* concept since distributed setups can be realized (cf. Section 6.4.2, p. 157), where simulations and the *Model-Testbed* are executed and placed at different hosts and locations.

To provide a power estimation early in the development process while preventing time-consuming *edit-cross-compile-flash-debug* cycles [30, 393], the software application itself is not executed on *Model-Testbeds* directly. One disadvantage of this approach is that computationally

intensive parts of the software model, for which no low-level source code exists at the time of testing, are not included in the power consumption estimation process. However, regarding the overall system, the MCU only accounts for a small part of the total consumption. Since *Model-Testbeds* can physically map different operating modes, only a small delta of the current consumption in the active state is not considered. Again, the use of concepts such as scenarios (cf. Section 3.2, p. 81 ff.) is a possible solution in which the total workload for the active state of an MCU within test cases may be specified. With adaptations of *Model-RPC* and the firmware, the *Model-Testbed* might be able to simulate such workloads.

6.5.2 Software Layer

By implementing the *Model-RPC* protocol for message exchange, the firmware executed on the *Model-Testbed* provides a universal interface that does not require a specific system or simulation environment. Applications such as the UC^{2}E tool can send requests to the *Model-Testbed*, which are processed and directed to the desired hardware component, e.g., the MCU, or communication interfaces such as I^{2}C, SPI, or UART. In this context, the *Model-Testbed* acts as a server for requests from any client, e.g., one or more MDD tools. The *Model-RPC* allows such clients to configure the *Model-Testbed* and to interact with its interfaces in a uni- and bidirectional manner.

Related work presented in Section 2.7.6 (p. 63 ff.) focuses on application-level data exchange between distributed co-simulations. In contrast, the concept of *Model-Testbeds* aims to link the software model with the target platform during the simulations. Similar to DCP, a specific firmware is executed on the hardware platform. However, the firmware does not represent a simulation. Instead of exchanging data at the application level, the firmware provides low-level access to the underlying hardware in a unified manner. This enables direct communication between the software model and hardware components.

In the following, the architecture of the firmware is presented. Furthermore, power modes are introduced which are realized by the firmware (message interpreter) of *Model-Testbeds* and the policy-oriented HAL presented in Section 6.3 (p. 149 ff.) to keep the system model platform-independent during simulation and to be able to support multiple MCUs from different vendors without adjusting parts of the system model.

Architecture

The firmware has been developed for each hardware platform presented in Section 6.5.3 (p. 165). Naturally, the basic principle of the firmware has to be adapted for the specific hardware platform and, therefore, contains platform-specific source code, e.g., to configure and interact with communication interfaces. However, all firmware variants are based on *FreeRTOS* [13].

Figure 6.17 shows a UML class diagram of the firmware where some aspects have been simplified to maintain readability. The architecture of the firmware consists of five central classes. The main class of the firmware, referred to as `MainApp` in Figure 6.17, manages all `MessageDispatcher` and `MessageHandler` instances executed in distinct tasks. The firmware provides two `MessageDispatcher` implementations to handle all incoming and outgoing *Model-RPC* messages, namely the `ConfigurationDispatcher` and the `ControlDispatcher`. While the `ConfigurationDispatcher` accepts all *Model-RPC*-related messages for the configuration of the *Model-Testbed*, the `ControlDispatcher` handles all requests for the system and for

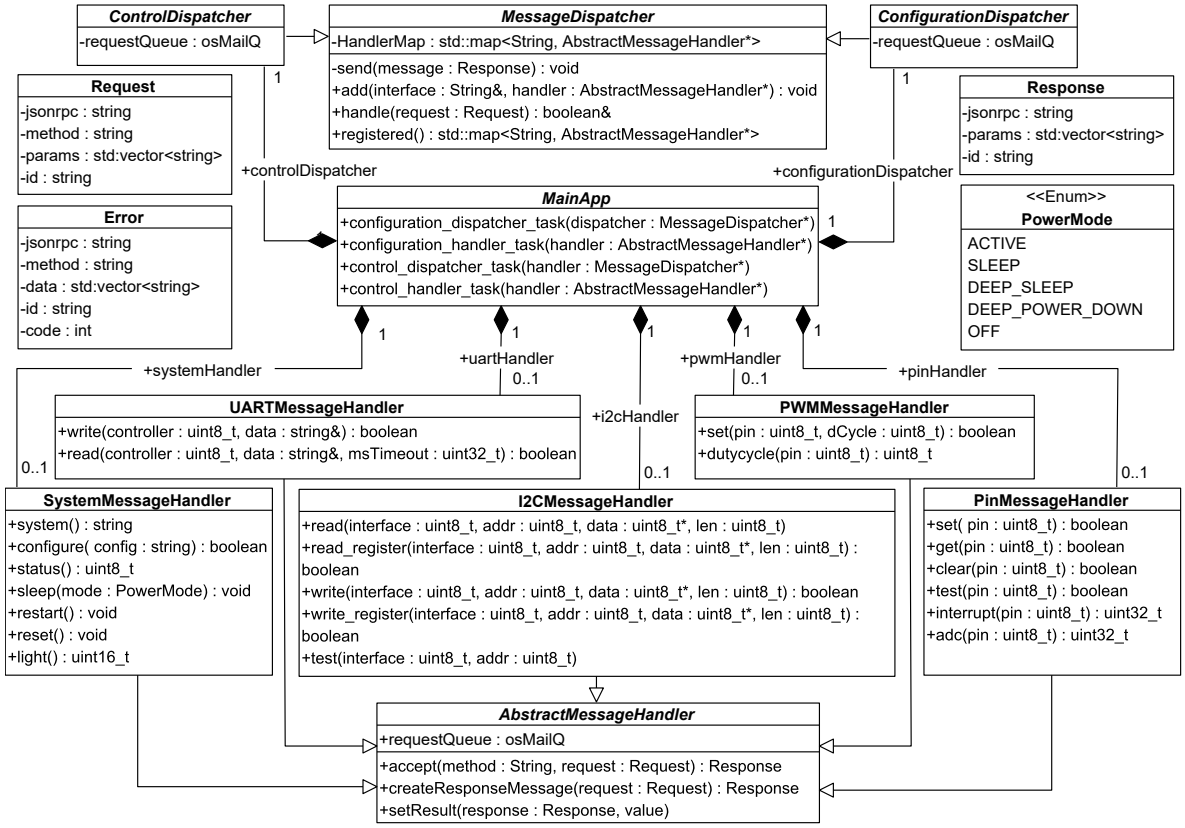


Figure 6.17: Basic architecture of the firmware (message interpreter), adapted from [62] (UML 2.5 class diagram notation).

communication interfaces to which peripheral devices may be connected and distributes the request to the appropriate handler instance. As Figure 6.17 further illustrates, a platform-specific handler implementation has been developed for each communication interface which executes the request and, if necessary, generates a response, which is processed by the `ControlDispatcher` instance. Classes for the message data types, such as `Request`, `Response`, and `Error`, follow the *Model-RPC* structure.

The current version of the firmware does not support multiple responses for a single request and the initiation of *Model-RPC* messages if interrupts are triggered. However, when GPIOs are configured as interrupts, the firmware provides functions to query an internal interrupt counter for each configured GPIO. By this, external clients may use a polling approach to detect if an interrupt has occurred.

Realization of Power Modes

On one hand, the *Model-Testbed* enables functional access to the connected devices. On the other hand, it is possible to change the system state of the *Model-Testbed*, e.g., by switching the power mode of the MCU. Due to different architectures, vendor-specific design choices, and features, MCU families provide various power modes with different specifications that define whether CPU cores, flash units, SRAM banks, oscillators, and interfaces are powered,

turned off, or throttled if possible. Due to this, a uniform control of MCUs requires further abstraction. However, while each vendor may use a different naming scheme for provided power modes, they might share similar characteristics. The basic idea is to define a set of abstract power modes so that hardware component models representing MCUs in the model domain may provide a uniform interface for controlling power-related behavior. These power modes are implemented by the firmware of *Model-Testbeds* (cf. `PowerMode` in Figure 6.17) and the policy-oriented HAL (cf. Figure 5.2, p. 123) to ease the simulation of UML models and support the rapid prototyping approach. Within this thesis, a total of five power modes have been defined, namely:

- *ACTIVE*: Defines the state where the MCU is active and all peripherals powered.
- *SLEEP*: The system clock is stopped, and no instructions are executed. Peripherals are powered and can generate interrupts. If configured, registers, the DMA, GPIOs, static RAM and flash modules are maintained.
- *DEEP_SLEEP*: This mode is similar to the *SLEEP* mode with the main clock and peripheral clocks disabled. Flash modules are put in standby mode or turned off.
- *DEEP_POWER_DOWN*: Except for the *Real-time Clock (RTC)*, the MCU is completely turned off and can only be awakened by RTC-generated interrupts. Static RAM and registers are not maintained.
- *OFF*: The MCU is completely turned off.

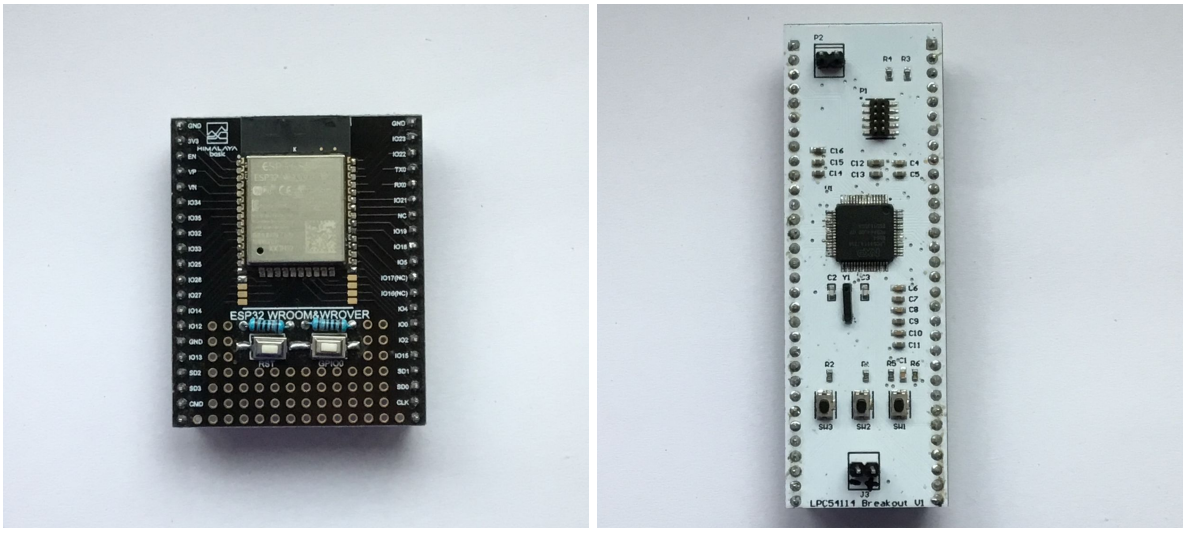
Unique ids are assigned to the introduced power modes, ranging from 0 (*ACTIVE*) to 4 (*OFF*), which are used as a reference by the firmware and *Model-RPC*. For each MCU of a *Model-Testbed*, the introduced abstract power modes have to be mapped to the most suitable power modes of the specific MCU. The elaborated mappings for the MCUs discussed in Section 6.5.3 are shown in Table E.1 as part of Appendix E (p. 285 ff.).

6.5.3 Hardware Layer

To demonstrate the portability of the approach, *Model-Testbeds* were developed for three different hardware platforms. The MCUs used for *Model-Testbeds* were chosen for a variety of reasons. On one hand, the platform independence of the approach shall be demonstrated by selecting products with various architectures from different manufacturers. On the other hand, the selected MCUs have different fields of application. For instance, the Espressif Systems ESP32 [111] is a widely used MCU in the IoT domain, applied in commercial products, e.g., smart home applications, and in IoT-related projects driven by the maker community. The NXP LPC54114 [269] is a low-power, dual-core MCU based on an ARM Cortex-M4 core and an ARM Cortex-M0+ core, specifically designed for always-on applications in automotive, industrial, and mobile domains. The STMicroelectronics STM32 MCU family is widely used in the industry for electronic devices focused on sensor data handling and data processing. Based on the provided functionalities, *Model-Testbeds* may be further divided into basic and advanced *Model-Testbeds*.

Basic Model-Testbeds

The term *basic* describes platforms following the concept of breakout boards. Figure 6.18 shows *Model-Testbeds* which have been used and developed within this thesis. The first *Model-Testbed* shown in part (a) of Figure 6.18 is built on a Himalaya basic breakout board with a mounted Espressif ESP32 SoC[2] without embedded flash. A second *Model-Testbed* based on the NXP LPC54114[3] [269] is shown in part (b) of Figure 6.18. To meet the design concept of *Model-Testbeds*, a specific breakout board for the NXP LPC54114 has been designed as a platform prototype using the Altium CircuitMaker design tool [12]. The schematics of the developed breakout board are included in Appendix E.2 (p. 287 f.).



(a) (b)

Figure 6.18: Images of basic *Model-Testbeds* based on a commercial ESP32 breakout board (a) and the developed breakout board for the LPC54114 (b).

Advanced Model-Testbed

An advanced *Model-Testbed* based on the STMicroelectronics STM32L476 [371] as an improved version of the basic *Model-Testbeds* has been elaborated in a supervised master's thesis [62]. The main goal of the advanced *Model-Testbed* concept is to enhance the rapid prototyping support and to improve the usability, productivity, and experience of developers by simplifying their general use and handling for test setups. For instance, when basic *Model-Testbeds* are used, system setups were based on breadboards and jumper cables, which has turned out to be occasionally error-prone and required more in-depth technical knowledge. Due to this, the mobility of *Model-Testbeds* was also limited.

The developed advanced *Model-Testbed* addresses those limitations and is designed as a mobile, stackable, and enclosed system with standardized *JST XH* pin and socket interconnection for communication interfaces such as GPIO, I^2C, or SPI. Figure 6.19 shows the

[2]Espressif part number: ESP32-WROOM-32 with ESP32-D0WDQ6
[3]NXP part number: LPC54114J256BD64QL

(a) (b)

Figure 6.19: Images of the advanced *Model-Testbed* based on the STM32L476 showing the front (a) and side (b) of the 3D-printed enclosure.

front (a) and side (b) of the developed advanced *Model-Testbed*. It provides eight digital and four analog interfaces via shared GPIOs and four *Pulse-width Modulation (PWM)*, two I^2C, and two UART interfaces. Moreover, the advanced *Model-Testbed* provides a single interface for CAN and SPI, respectively. Furthermore, two buttons and a brightness sensor for direct interaction with developers have been integrated. A separate non-volatile memory module stores the active configuration and up to three fallback configurations. The simplified block diagram of the *Model-Testbed* is shown in Figure 6.20. Besides the STM32L476, a Raspberry Pi Zero W [318] has been integrated into the design to provide a wireless access point with which developers are able to configure the *Model-Testbed* using a web user interface.

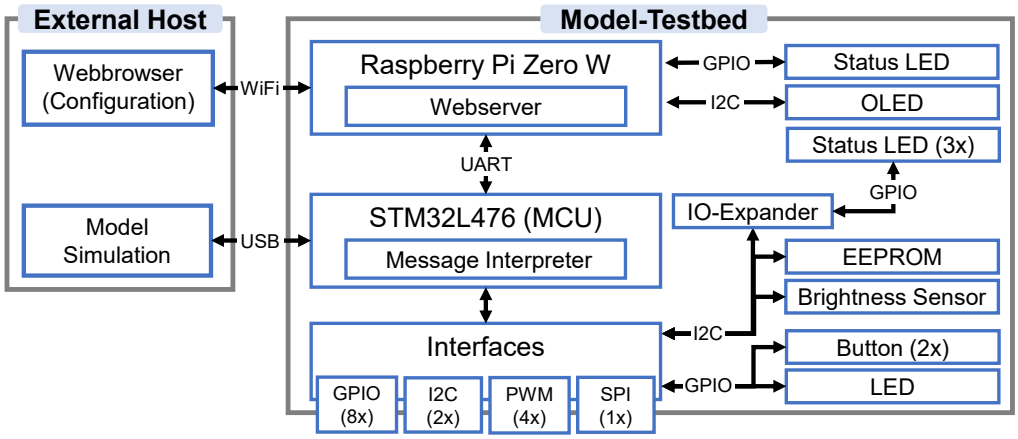


Figure 6.20: Block diagram of the advanced *Model-Testbed* (adapted from [62]). Rectangles outlined in grey define the system boundaries, while rectangles outlined in blue describe the individual components. Black arrows indicate communication protocols between components.

6.5.4 Model-RPC

To enable interactions between the simulation environment and a *Model-Testbed* within DPA (cf. Section 5.4.2, p. 136), *Model-RPC* as a universal and lightweight communication protocol has been specified, marked as (6) in Figure 6.1 (p. 139). Generally, *Model-RPC* can be understood as a mapping of functions a *Model-Testbed* provides. In DPA, the interaction between the simulated system model and the *Model-Testbed* follows a client-server principle and relies on RPC to exchange messages. As illustrated in Figure 6.1 (p. 139), if a hardware component model as the *client* requires hardware access, corresponding SDXP messages are translated into *Model-RPC* messages by the UC^{2}E tool. Afterward, the messages are forwarded to the *Model-Testbed*, acting as a *server* in this scenario. The responses are returned from the *Model-Testbed* to the UC^{2}E tool and are passed to the simulation environment for further processing. Due to the strict decoupling, *Model-RPC* fulfills the following requirements:

- *Independent*: The communication protocol should be independent of the transport protocol, communication interface, hardware platform variants, and programming languages.
- *Efficient*: The implementation of the endpoints should be resource efficient w.r.t. the source code footprint and the required computing time to interpret messages.
- *Structured*: Messages should be structured so that they are machine-readable.
- *Verifiable*: Along with the machine-readable structure of messages, a methodology should exist to verify messages automatically for correctness and completeness using a schema description.
- *Extendible*: The structure of messages should support future extensions.
- *Versionable*: The communication protocol should offer the possibility to distinguish between different versions to avoid problems with varying endpoints.
- *Well-defined*: The communication protocol should follow a standardized approach to easily integrate and adapt existing tools and procedures.

Data serialization, as used for RPC, can be in textual or binary format. Flatbuffers [141] and Protocol Buffers [139] are examples of language and platform-independent mechanisms for a binary serialization of structured data. With gRPC [140], a standardized framework for the binary message exchange based on RPC and Protocol Buffers exists. XML and JSON are the two most commonly used data exchange formats for text-based encoding of messages. While XML is the approach with the most comprehensive capabilities, processing XML-based messages on an embedded system is too complex compared to JSON [266].

Based on and fully compatible with JSON-RPC [190], *Model-RPC* is defined as a standardized communication and human-readable protocol. The disadvantages of lower performance and higher amount of data compared to binary encoded messages [233, 310, 373] are accepted for the prototype implementation within this thesis. JSON-RPC implements a request-response principle based on a client-server model with JSON-based messages. The specification [190] defines four message types: (1) `request`, (2) `response`, (3) `notification`, and (4) `error`. A client initiates a `request` message while the server sends a `response` message back to the client containing either the result or an error message. A client may also send a `notification` message. In this case, the server does not send a `response` message, even if the execution

fails. In addition to the single message mode, several messages can be sent as a batch. The basic structure of a *Model-RPC* request is shown in Listing 6.8.

```
1 {
2    "jsonrpc": "2.0",
3    "method": "[CLASS_NAME].[METHOD]",
4    "params": {},
5    "id": 12
6 }
```

Listing 6.8: Basic structure of a *Model-RPC* request message.

The message structure consists of a `method` property as a string, a `param` object, and an `id`, which the client can choose freely to assign a response to a former request. A *Model-RPC* `request` reuses the `jsonrpc` property (line 2) to specify the version number. For all messages, this property is set to `2.0`, representing the latest version of JSON-RPC [190]. However, the structure of the `method` property value (line 3) is more restricted compared to JSON-RPC and consists of the two parts `[CLASS_NAME]` and `[METHOD]` separated by a dot. The class name specifies a part of the *Model-Testbed*, whereas the method defines the actual operation performed by the part. This extension expands the language syntax in an object-oriented way to distinguish between different component types. It also allows commands and functions for specific interfaces to be grouped and organized, achieving a logical or physical segmentation of the *Model-Testbed*. The `params` object (line 4) may contain an array or any structured object. In *Model-RPC*, a `device` object must be provided as a parameter to distinguish between communication interfaces of the same type, e.g., multiple I^2C buses. The response message shown in Listing 6.9 contains the result of the method execution as a `result` property, e.g., `42`. The format of the result property may vary between a simple value or a JSON object literal. In Listing 6.9, the value of the `id` property matches the value of the `id` property in the former request message (cf. Listing 6.8). Note that a `response` message will only be sent to the client if the request contains an `id` property.

```
1 {
2    "jsonrpc": "2.0",
3    "result": 42,
4    "id": 12
5 }
```

Listing 6.9: Exemplary *Model-RPC* response message.

Model-RPC provides different method classes to configure, control, and interact with the *Model-Testbed* and interfaces such as GPIO, PWM, UART, and I^2C on a lower level. The remainder of this section describes the specified method classes (`[CLASS_NAME]`) with all provided methods (`[METHOD]`) in detail.

System Methods

The `system` class identifier groups all methods addressing the configuration of the *Model-Testbed* and the behavior of the MCU. Table 6.5 shows the set of methods provided by the `system` object identifier.

Method	`params` Parameter		Description
	Name	*Type*	
`system.config`	-	-	Return current configuration.
`system.restart`	-	-	Restart remote target.
`system.reset`	-	-	Reset remote target.
`system.sleep`	mode	Number	Set the *Model-Testbed* into a specific low-power mode.
`system.configure`	config	`configType`	Configure a remote target.

Table 6.5: *Model-RPC* methods for the `system` class.

With the `system` class identifier, the target *Model-Testbed* can be reset, restarted, and set to a specific low-power mode. For the latter, a `mode` parameter must be provided which corresponds to the id of the respective low-power mode specified in Section 6.5 (p. 161 ff.). The example in Listing 6.10 shows a *Model-RPC* request to set the target system into a low-power mode, where the `mode` parameter is added as property of the `params` object (line 4) and set to id 1 (sleep mode). With the `config` and `configure` methods, the current setting of the *Model-Testbed* can be read and set using a `configType` object. The structure of the `configType` object contains configuration details for each supported interface of the *Model-Testbed*, such as GPIO, PWM, or I^2C. It is added as a property to the `params` object (line 4) if the `configure` method is used. More details about the `configType` object, along with JSON-Schema [417] description, are provided in Appendix D.1 (p. 279 f.).

```
1  {
2     "jsonrpc": "2.0",
3     "method": "system.sleep",
4     "params": { "mode": 1 }
5  }
```

Listing 6.10: Basic example of a *Model-RPC* to change the power mode of a *Model-Testbed*.

GPIO Methods

To control a single GPIO, *Model-RPC* specifies the `pin` class identifier with the implemented methods shown in Table 6.6. In order to interact with a specific GPIO, the `params` object is extended with a `device` object containing a key-value property with the number of the affected GPIO. In *Model-RPC*, a GPIO with a default configuration can be set, cleared, and read. However, if a GPIO is configured as ADC or interrupt, extended functionalities are provided by the firmware of a *Model-Testbed* (cf. Section 6.5.2, p. 163 ff.). With `pin.adc`, the analog value of a GPIO can be requested, while `pin.interrupt`, as a counter-like method, returns the number of interrupts triggered. The following example in Listing 6.11 shows a

Method	`device` Parameter		Description
	Name	*Type*	
`pin.set`	gpio	Number	Set a single GPIO.
`pin.clear`	gpio	Number	Reset a single GPIO.
`pin.get`	gpio	Number	Get the current state of the GPIO.
`pin.adc`	gpio	Number	Get the current ADC value from 0–4048, e.g., 0–3.3 V.
`pin.interrupt`	gpio	Number	Get the counted number of rising/falling edges for a GPIO configured as interrupt pin.

Table 6.6: *Model-RPC* methods for the `pin` class.

Model-RPC request to receive the current analog value of the GPIO with the number 4. Note that an `id` is provided to receive a response with the current value from the *Model-Testbed*.

```
{
   "jsonrpc": "2.0",
   "id": 1,
   "method": "pin.adc",
   "params": {
      "device": {
         "gpio": 4
      }
   }
}
```

Listing 6.11: Basic example of a *Model-RPC* request to get an analog value of a single GPIO.

PWM Methods

Model-RPC specifies the PWM class identifier to provide functions of the PWM interface to read and configure the duty cycle for a specific PWM channel. An overview of provided methods is given in Table 6.7.

Method	`device` Parameter		Description
	Name	*Type*	
`pwm.dutycycle`	pwm	Number	Get the duty cycle of the PWM channel.
`pwm.set`	pwm dutycycle	Number Number	Set the duty cycle for a PWM channel.

Table 6.7: *Model-RPC* methods for the `pwm` class.

UART Methods

Model-RPC provides `write` and `read` methods for peripheral devices connected via UART interfaces. Both methods listed in Table 6.8 require an id of the UART `controller` as a

property of the `device` object. Additionally, a `buffer` containing the string to be sent must be provided for the `write` method. The `read` method requires a `timeout` property defining the amount of time the method may take to complete. The key-value pairs `buffer` and `timeout` in Table 6.8 have to be provided as properties of the `params` object.

Method	**Parameter**		**Description**
	Name	*Type*	
`uart.write`	controller*	Number	Write a message to an UART interface.
	buffer+	String	
`uart.read`	controller*	Number	Read data from an UART interface.
	timeout+	Number	

* = `device` object property, + = `param` object property

Table 6.8: *Model-RPC* methods for the `uart` class.

Figure 6.21 shows an example for a communication between a system model and a *Model-Testbed* to send and verify a configuration for a peripheral device connected over UART using `notification`, `request`, and `response` messages. In step 1 of the UML sequence diagram, a *Model-RPC* notification message with `uart.write` as method is transmitted from the simulated system model to the *Model-Testbed*. The message includes a `device` object and a `buffer` object as key-value pair of the `params` object. While the `controller` key-value pair specifies the UART interface, the `buffer` contains the configuration, which will be written to the connected peripheral device on the *Model-Testbed*. In step 2.1 of Figure 6.21, the simulated system model sends a *Model-RPC* request message to to delegate the execution of a

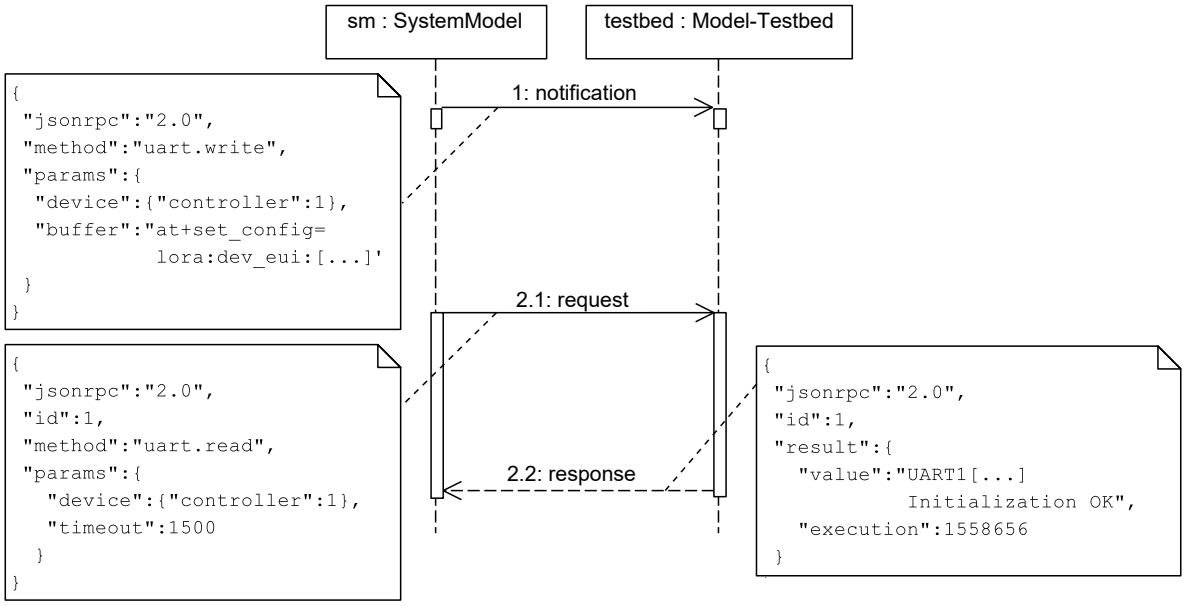


Figure 6.21: *Model-RPC* communication between a system model and a *Model-Testbed*, adapted from [341] (UML 2.5 sequence diagram notation).

`uart.read` command on the target *Model-Testbed*. Since the request message contains an `id` parameter, the *Model-Testbed* generates a response message that is sent back to the system model in step 2.2. The `value` parameter of the result contains the response of the peripheral device, which the system model uses to verify if the configuration has been processed correctly.

I²C Methods

Model-RPC provides a set of methods listed in Table 6.9 for the interaction with I²C interfaces. The `read` method requires the `param` object to contain a key-value pair for the number of `bytes` to be read. The `device` object must consist of properties as key-value pairs for the I²C `bus` to be used, the `identifier` of the connected device, and the `register` to be read. For some peripheral devices, the `register` property for `read` methods can be omitted. The `write` method requires a list of bytes to be written and a `device` object with properties similar to those of the `read` method. Furthermore, a basic `test` function to verify the presence of a peripheral device with the given `identifier` on a specific I²C `bus` is provided.

Method	**Parameter**		**Description**
	Name	*Type*	
`i2c.read`	bytes^{+}	Number	Read bytes from a I²C device.
	bus*	Number	
	identifier*	Number	
	register*	Number	
`i2c.write`	bytes^{+}	List	Send bytes to a I²C device.
	bus*	Number	
	identifier*	Number	
	register*	Number	
`i2c.test`	bus*	Number	Checks the presence of a peripheral device with the given `identifier`.
	identifier*	Number	

* = `device` object property, $^{+}$ = `param` object property

Table 6.9: *Model-RPC* methods for the `i2c` class.

The *Model-RPC* definition can be fully described using OpenRPC [283]. The OpenRPC specification provides a machine-readable and programming language-agnostic interface description for JSON-RPC APIs allowing humans and computer systems to explore the structure and methods. An example of the *Model-RPC* definition for the `uart.write` method in OpenRPC can be found in Appendix D.2 (p. 283 f.).

In this chapter, the prototype implementation of the DPA method has been presented. This includes parts (1) to (8) illustrated in Figure 6.1 (p. 139), for instance, the concept of *Model-Testbeds*, the SDXP and *Model-RPC* communication protocols, and the UC²E tool as a central element combining, managing, and controlling UML-based simulations, *Model-Testbeds*, and measuring devices. Hence, a power consumption estimation can be executed, as described in steps 6 to 11 of the developer workflow (cf. Section 3.1, p. 77 ff.).

Chapter 7

Evaluation

After discussing the main concepts and their prototype implementations, this chapter evaluates the overall power consumption estimation approach as a contribution to answer RQ3 and RQ4. Parts of this chapter have been published in [341]. Section 7.1 introduces the basic setup of the case study for the power consumption estimation. The evaluation is divided into two parts. As the first part, Section 7.2 presents the modeling and evaluation of the beehive microclimate sensor node case study to demonstrate the collaboration of the presented concepts introduced in this thesis and the potential of the overall power consumption estimation approach. Furthermore, the potential of the power consumption estimation approach for the detection of energy bugs is shown. Section 7.3 focuses on the performance of DPA and evaluates the time delays and power overhead as the second part of the evaluation.

7.1 Setup

This section introduces the basic test setup for the power consumption estimation of a beehive microclimate sensor node as part of the case study presented in Section 7.2 (p. 176 ff.). With the Espressif Systems ESP32, the selected *Model-Testbed* is based on a popular low-cost MCU commonly used in the IoT domain. The firmware executed on the *Model-Testbed* relies on FreeRTOS [13] and has been developed with the Espressif IoT development framework. The test setup shown in Figure 7.1 consists of the Qoitech Otii Arc measurement unit, the *Model-Testbed*, and additional hardware components used in the beehive microclimate sensor node, namely the Bosch BME280 to measure environmental parameters and the RAKwireless RAK811 *Long Range (LoRa)* module to achieve a wireless communication. The measuring device in Figure 7.1 is connected to the host system executing IBM Rhapsody and the UC^{2}E tool via USB. A Silicon Labs CP2102 USB-to-UART bridge [359] with a configured baud rate of 115200 bauds is used to communicate with the *Model-Testbed*.

For the evaluation, the Qoitech Otii Arc measures the electric current consumption and voltage of the system shown in Figure 7.1 with a sample rate of 4 ksps. The Qoitech Otii Arc uses two shunts measuring in parallel, allowing the device to immediately switch between these two shunts to avoid the range problems and sample loss described in Section 2.1.3 (p. 20 ff.). The analyzes of the UC^{2}E tool are based on hardware component models and the raw data recorded by the measuring device and obtained at a polling interval of 1 ms from the Otii desktop application.

Due to the setup, the representation of signals in the diagrams of this section may be inaccurate if the frequency of the signals to be sampled is equal to or higher than the sampling frequency of the Qoitech Otii Arc or the polling interval of the UC^{2}E tool. Further details about the UC^{2}E tool and embedded firmware, hardware characteristics of the host system, and the Qoitech Otii Arc can be found in Appendix F (p. 289 ff.).

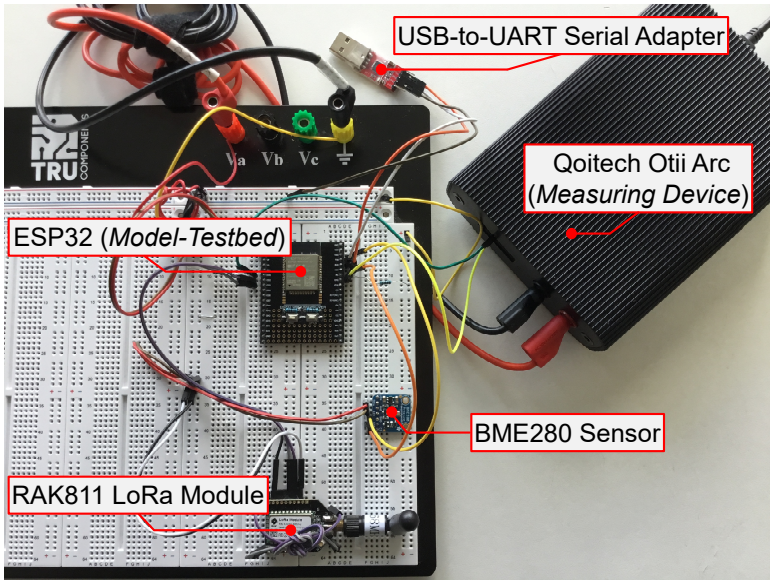


Figure 7.1: Test setup showing the *Model-Testbed* with connected peripherals and the Qoitech Otii Arc.

7.2 Case Study: Beehive Microclimate Sensor Node

This section introduces the beehive microclimate sensor node case study as a real-world IoT example for smart objects. The evaluation in this section demonstrates the application of the presented modeling concepts. Based on the case study, a proof-of-concept and an evaluation of the DPA methods are presented. Previous versions of the beehive microclimate sensor node case study have been used in [341, 391] to illustrate concepts and perform evaluations.

Section 7.2.1 provides an overview of the beehive microclimate sensor node, introduces the hardware components of the platform, and explains the field of application. The modeling of hardware components is covered by Section 7.2.2, while Section 7.2.3 introduces the software application model. In Section 7.2.4, the scenario of the case study is introduced. A power consumption estimation for the beehive microclimate sensor node is demonstrated and discussed in Section 7.2.5, while the capabilities of DPA to detect energy bugs are shown in Section 7.2.6.

7.2.1 Overview

This section describes the basic layout and functionality of the beehive microclimate sensor node to monitor western honeybee (Latin: Apis mellifera) colonies as a typical IoT application example. Generally, beekeeping in wood or polystyrene magazines can negatively affect bees

since the magazines do not correspond to natural housing. Bees can actively influence the humidity and temperature of a magazine to ensure optimal environmental conditions for larvae and the colony [106]. Since beekeepers either do not visit the hives at all or visit them irregularly during the winter months from about October to March, the sensor node monitors the hives remotely. This results in the requirement that the sensor node must operate energy-efficiently for at least six months while monitoring the actual condition in the magazine and transferring environmental data to a cloud service via a wireless communication interface.

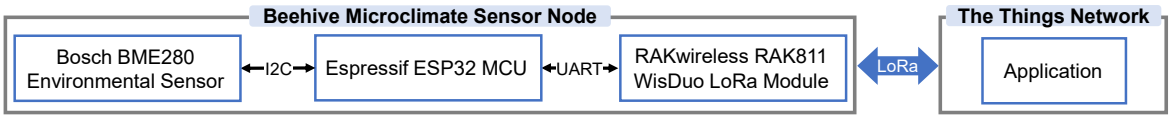


Figure 7.2: Block diagram of the sensor node connected to the *The Things Network (TTN)*. Black arrows describe wired communication between components (blue squares), and the blue arrow describes wireless communication between systems (grey squares).

A simplified block diagram of the sensor node is shown in Figure 7.2. It consists of an Espressif ESP32 MCU [111], a Bosch BME280 environmental sensor [53] on an Adafruit breakout board [5] connected via I^2C, and a RAKwireless RAK811 WisDuo LoRa module [317] based on the Semtech SX1276 LoRa transceiver [351] connected via UART. The Bosch BME280 sensor is placed inside the beehive magazine and measures temperature, humidity, and air pressure at a fixed interval. For energy-efficient and long-range wireless communication, the LoRaWAN communication protocol [230] is used to transmit average values of the obtained sensor data to a TTN [385] cloud application instance, where the data may be pre-processed, formatted, and forwarded to platform services such as ThingSpeak [381] or tools like MathWorks MATLAB [380] for further analysis. The TTN is an IoT ecosystem based on an open and decentralized LoRaWAN network. Figure 7.3 shows the IoT sensor node prototype in a green-colored enclosure applied in a beehive.

Figure 7.3: The prototype microclimate sensor node applied in a beehive (red circle).

7.2.2 Hardware Component Modeling

This section describes the UML modeling process for each hardware component of the beehive microclimate sensor node following the developed concepts introduced in Section 5.2 (p. 117 ff.). Hardware component models representing peripheral devices use the reference implementation of the policy-oriented HAL (cf. Section 6.3, p. 149 ff.) for hardware-software interactions with the *Model-Testbed* (cf. Section 6.4.2, p. 157 f.). By this, hardware-related aspects are hidden from the actual software application model, increasing platform independence.

Device	**State**	**Electric Current**			**Execution Time**		
		Value	*Static*	*Source*	*Value*	*Static*	*Source*
Espressif ESP32	Off	1.0 µA	Y	O	-	-	-
	Deep-sleep	10.0 µA	Y	O	-	-	-
	Active	27.5 mA	Y	M	-	-	-
Bosch BME280	Sleep	13.4 µA	Y	M	-	-	-
	Forced	467.2 µA	Y	M	16.59 ms	Y	M
RAK-wireless RAK811	Sleep	5.5 µA	Y	M	-	-	-
	Idle	6.1 mA	Y	M	-	-	-
	Join	19.9 mA	Y	M	-	-	-
	Receive (RX)	22.2 mA	Y	M	1300 ms	Y	E
	Transmit (TX)	63.9 mA	Y	M	D	N	C

- = Not applicable, D = Dynamic at runtime, C = Calculated, E = Estimated,
M = Measured, N = No, O = Obtained from datasheets, Y = Yes

Table 7.1: Operating states, power consumption and execution times of components for the beehive microclimate sensor node. Unused operating states have been omitted.

To define an energy model for each hardware component model, datasheets [53, 111, 317] for the components of the beehive microclimate sensor node shown in Figure 7.2 (p. 177) have been analyzed, and corresponding operating states have been extracted. Table 7.1 shows a subset of identified operating states which can be triggered by the software application model (cf. Section 7.2.3, p. 183 ff.), along with their initially obtained current consumption and execution time values. The supply voltage of each hardware component in Table 7.1 is 3.3 V. For the evaluation, transitions are assumed to be executed instantaneously and do not contain any significant energy current or time offset. As a result, the tagged values `current` and `execTime` of each transition are set to $(0, mA)$ and $(0, ms)$, respectively. The electric current consumption values in datasheets are often determined under ideal conditions and differ significantly from those reached when the hardware component is used in more realistic environments. Therefore, for all considered operating states, the extracted power consumption values have been confirmed or adjusted with values obtained by measurements executed in this thesis under more realistic conditions of the environment in which the evaluation is performed. In the following, particular aspects of the hardware modeling process are discussed.

Espressif ESP32 MCU

For the Espressif ESP32 MCU, a power mode mapping (cf. Section 6.5.2, p. 163 ff.) for the operating states given in Table 7.1 has been defined. The `ACTIVE` state corresponds to the *Modem Sleep* power mode of the ESP32 (cf. Table E.1 in Appendix E.1, p. 285 ff.). The electric

current consumption value for this state has been captured while the MCU was operating in a mode with a single active core and disabled Wi-Fi and Bluetooth peripherals. Within the `DEEP_SLEEP` state, mapped to the `Deep-sleep` [111], only the ultra-low power core, the real-time clock module, and the memory are powered. The current consumption value for the `OFF` state has been obtained from the datasheet [111], while the values for the two other states, `DEEP_SLEEP` and `ACTIVE`, have been derived from measurements. Since the time spent in each mentioned state during the simulation depends entirely on the workflow of the software application model, no execution times have been specified.

Bosch BME280 Environmental Sensor

As a result of the modeling process, Figure 7.4 shows the UML class definition of the Bosch BME280 hardware component model directly taken from the IBM Rhapsody MDD tool. The hardware component model inherits from the `PeripheralDevice` base class as described in Section 5.2.3 (p. 122 ff.). Additionally, an interface (`BME280_Interface`) has been defined that specifies all functions a software application model can use. The hardware component model also encapsulates the platform-independent driver implementation [54] specified as externals files, `bme280` and `bme280_defs`, in Figure 7.4. Artifacts such as the official Bosch driver files annotated with the «`File`» stereotype in Figure 7.4 may be integrated using reverse engineering processes. The integration of legacy code is a typical procedure in MDD, independent of the presented concepts. It can be applied during the hardware modeling process described in the developer workflow (cf. Section 3.1, p. 77 ff.). Since the sensor is connected as an I^2C device, the hardware component model has a reference to an instance of a `SimulationI2CAccess` class as an implementation of the `RegisterAccessPolicy` introduced by the reference implementation of the policy-oriented HAL in Section 6.3 (p. 149 ff.).

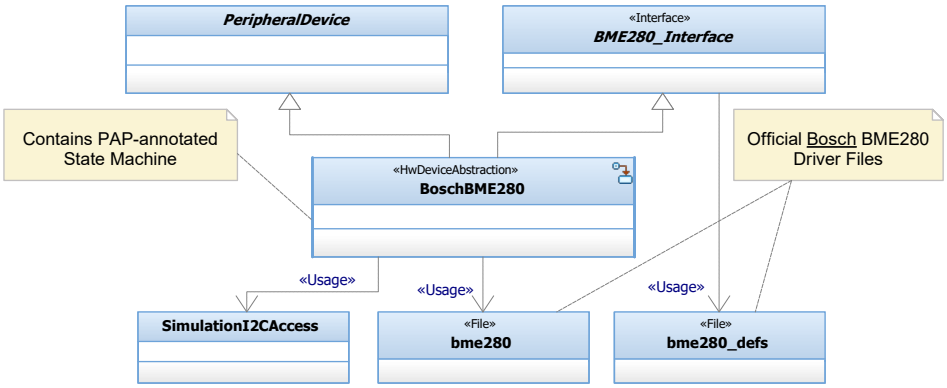


Figure 7.4: Bosch BME280 hardware component model as UML class diagram modeled with IBM Rhapsody. In order to improve the legibility, no attributes and operations are shown.

The state machine of the environmental sensor is illustrated in Figure 7.5. For the `Normal` state, the values for electric current consumption and the execution time are considered as being dynamic and specified by expressions, as a modeling feature of the PAP. The expressions are derived from the datasheet and depend solely on the current configuration, e.g., the number of sensors and the oversampling rate of each sensor during a single measurement. However, during the active phase of the case study, only a single measurement is obtained instead of

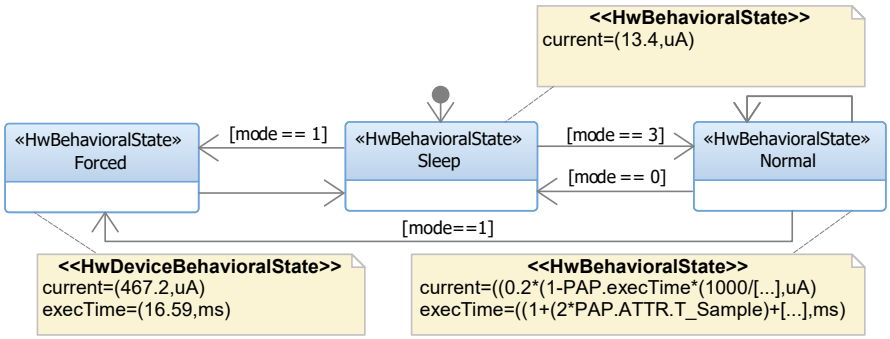


Figure 7.5: UML behavioral state machine of the Bosch BME280 modeled with IBM Rhapsody.

a (continuous) measurement series. Due to this, the sensor only switches between the `Sleep` and the `Forced` state during the simulation. Moreover, since the configuration is set by the software model in the initialization phase and does not change at runtime, the current consumption and execution time of the `Forced` state may be, instead, defined as static with fixed values, as shown in Figure 7.5 and Table 7.1 (p. 178).

RAKwireless RAK811 WisDuo LoRa Module

The state machine of the RAKwireless RAK811 LoRa module shown in Figure 7.6 is the most complex state machine of the microclimate sensor node with the highest uncertainties of the modeled values. The main reason is that the RAK811 module operates as a black box and does not provide internal indicators to identify the current operating state. As a

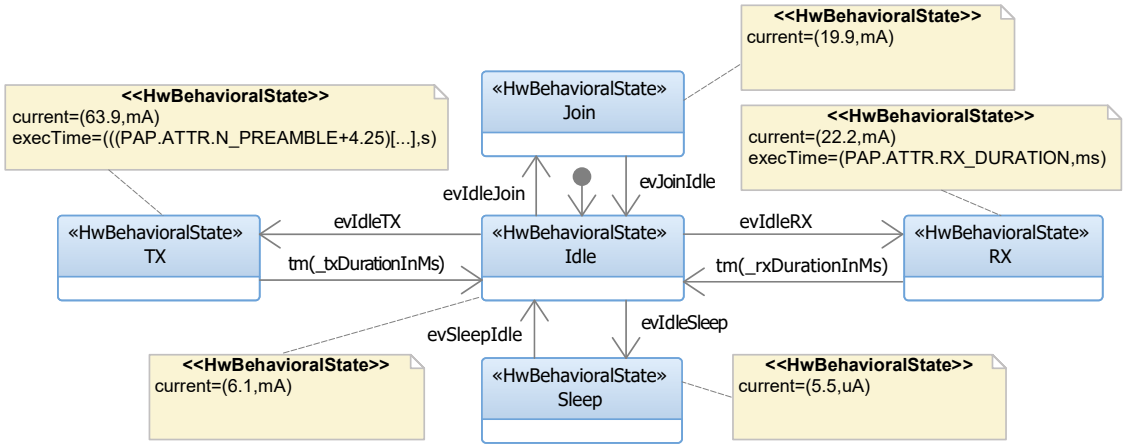


Figure 7.6: UML behavioral state machine of the RAK811 modeled with IBM Rhapsody.

result, driver implementations have to use polling mechanisms to verify whether, e.g., the *join* process [230] or single transmissions were processed successfully. Power consumption values for each operating state are obtained with the parameters described in Table 7.2 as the configuration for the wireless communication of the RAK811 LoRa module. The timer function `tm(...)` is applied on transitions in Figure 7.6 and defines a timeout, after which the *Receive (RX)* and *Transmit (TX)* windows are expected to be completed, and transitions

back to the `Idle` state are performed. The parameters of this method are either calculated by the hardware component model or defined by the scenario specifications. Events for a state transition are generated within states and defined with the prefix `ev`. The execution time of the TX window is calculated dynamically (denoted as D in Table 7.1) during simulation based on the parameters shown in Table 7.2, while the message length (in bytes) depends on the input of the software application model. The equations to calculate the length of the TX window are obtained from [351] and combined into a single expression used in the value field of the `current` tag provided by the PAP.

Parameter	Configured Value
Bandwidth	125 kHz
Spreading Factor	12
Preamble Length	23 symbols
Header Mode	Explicit (`bool`: 0)
Low Data Rate Optimization	Off (`bool`: 0)
CRC Check	Off (`bool`: 0)
Coding Rate	4/5 (`bool`: 1)

Table 7.2: Configuration parameters for the wireless communication of the RAKwireless RAK811 hardware component model, e.g., for calculating the TX window size. A more detailed explanation of the parameters can be found in [351].

The parameters shown in Table 7.2 may vary during the execution of test cases and either be included in scenario definitions (cf. Section 3.2, p. 81 ff.) or integrated into the hardware component model using the PAP stereotype for attributes. When dynamic power-related behavior is modeled, such expressions may become complex, as shown in Listing 7.1 for the complete definition of the TX duration value. The green-colored element represents

```
(((PAP.ATTR.N_PREAMBLE+4.25)*(1/(PAP.ATTR.BW/Math.pow(2,PAP.ATTR.SF)))+
 (((8+
   Math.max(
     Math.round(
       Math.ceil((8*PAP.ATTR.PAYLOAD_BYTES-4*PAP.ATTR.SF+28+16*
         PAP.ATTR.CRC_ENABLE-20*PAP.ATTR.IH_ENABLE)/(4*(PAP.ATTR.SF
         -2*PAP.ATTR.LOW_DATA_RATE_ENABLE))
       )*(PAP.ATTR.CR+4)
     )
   ,0)
 )+4.25)*(1/(PAP.ATTR.BW/Math.pow(2,PAP.ATTR.SF))))
,s)
```

Listing 7.1: Expression for the variable execution time in the TX state of the RAK811.

the MARTE `TupleType` definition in the form `(value,unit)`. Elements highlighted orange reference to the `id` of class attributes with annotated `HwAttribute`-based stereotypes. Mathematical operations not part of the VSL specification and represented by JavaScript operations (cf. Section 6.4, p. 155 ff.) are colored in cyan.

The beehive microclimate sensor node is configured as LoRaWAN class A device [350], the most energy-efficient type defined in the LoRaWAN specification [230]. As the main

characteristic of class A devices, communication can only be initiated by the end device when opening a TX window to send an uplink message, as illustrated in Figure 7.7. After each uplink transmission from the SUT to a TTN gateway, the end device, e.g., the RAKwireless RAK811, must open two receive windows, *RX1* and *RX2*, where data can be sent from the network via a gateway, to the SUT. The second window *RX2* will only be opened if no downlink message has been received during the first RX window. The transmission is completed if a message has been received or the second RX window slot is closed. Afterward, the RAK811 LoRa module returns to an idle state. The delay between the TX and RX windows and the length of each RX window are variable and part of the LoRaWAN protocol definition used by the TTN to configure devices. According to the log output after the RAK811 has connected to the TTN IoT ecosystem, the length of both RX windows is expected to be 3000 ms or less. The first window *RX1* is opened 5000 ms after a completed transmission (*RX1 Delay*), while the *RX2* opens after 6000 ms (*RX2 Delay*) if no data has been received within *RX1*.

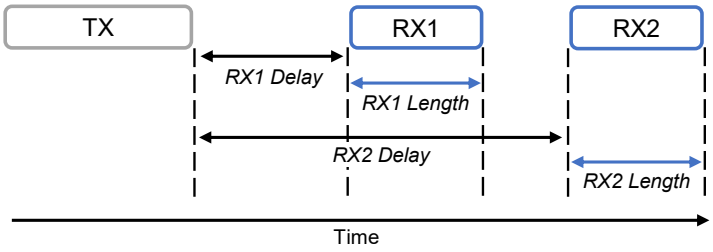


Figure 7.7: Receive windows and delays for LoRaWAN class A devices.

For accurate power estimations, the state machine of the RAK811 hardware component model has been further adapted to consider the characteristics of the TTN. First, an additional high-level state for the connection phase of the TTN network, also described as *Join*, has been added. Joining the TTN network is a non-deterministic process that may contain multiple attempts with various TX and RX windows, for instance, if a join accept message has been missed by the RAK811 module. Since the duration and power consumption can not be predicted in detail, an average power consumption value is used as long as the device tries to join the network. The value for the `join` state shown in Table 7.1 has been obtained by measuring the power consumption during this process ten times. Additionally, benchmarks revealed that the RAK811 LoRa module closed the first RX window always after a maximum time of 1500 ms, while the second RX window was never used, leading to parameter adaptions of the defined scenario and the state machine.

While the aforementioned aspects can be modeled for IPA and DPA using detailed scenario definitions, there are additional challenges when using DPA. The Bosch BME280, for example, only passively interacts with the environment by sensing the surroundings. However, variations in environmental properties directly impact the behavior of the RAK811 LoRa module. For instance, incoming RX messages as spontaneous events are not predictable by the UC^{2}E tool in real time. In addition, interference in the transmission medium can lead to longer transmissions and receiving times, which are unpredictable when interacting in real-time with physical hardware components in non-simulation environments. Since the RAK811 is configured as a class A device in this thesis, no spontaneous RX windows occur. Deviations during the power consumption estimation due to interferences of the transmission medium can be assigned to the states of the energy model and thus be identified as external effects.

This section has shown that the concepts of hardware component models introduced in Section 5.2 (p. 117 ff.) are able to represent the basic properties of all hardware components of the IoT sensor node case study in UML. With the concept of energy models, all significant power states can be considered, and both static and dynamic energy behavior can be modeled with stereotypes of the PAP (Section 5.3, p. 124 ff.), which contain single values and complex expressions. Furthermore, the novel variable declaration concept of the PAP used for attributes enables a connection between the functional and meta levels. While annotated attributes of a hardware component model may change during test case execution, e.g., by the software model, they may directly affect the computation of modeled expressions.

7.2.3 Software Application Modeling

This section describes the software application model for the beehive microclimate sensor node. From the software developer's point of view, the software application model represents the core element of the development process since it specifies the behavior of the sensor node based on the intended workflow and hardware-software interactions. As part of the system model, it is also the main target of the power consumption estimation process. Changes in the workflow may directly affect the system and lead to different measurement results. The UML class diagram of the system model developed in IBM Rhapsody is shown in Figure 7.8.

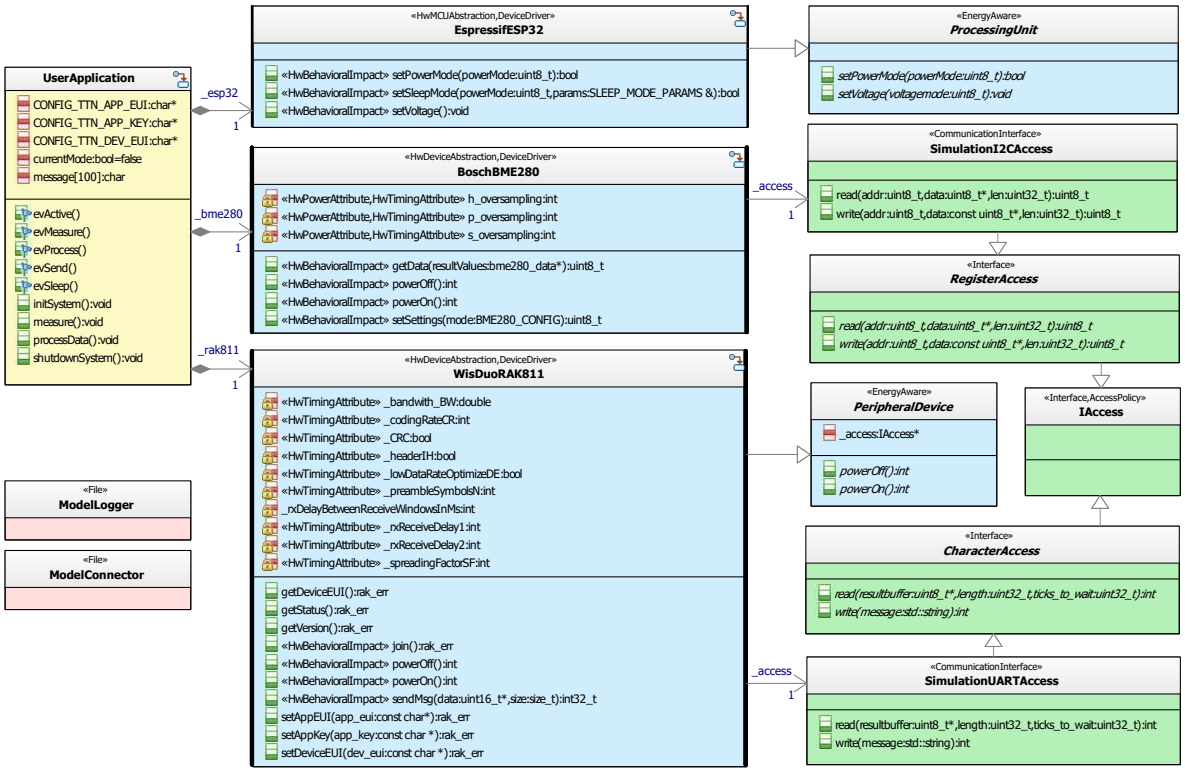


Figure 7.8: Class diagram of the beehive microclimate sensor node modeled in IBM Rhapsody, adapted from [341]. The system model consists of the main application (yellow), hardware component models (blue), and the policy-oriented HAL (green). Red-colored classes are extensions for data exchange with the UC^{2}E tool (UML 2.5 class diagram notation).

A basic software application has been developed to implement the functionality of the beehive microclimate sensor node. To maintain readability, only a subset of the defined attributes and operations are shown in Figure 7.8. The central part of the software application is represented by the yellow colored class `UserApplication` and defines the logic and workflow of the overall application. The blue highlighted classes show the hardware component models, namely `EspressifESP32`, `BoschBME280`, and `WisDuoRAK811`, along with their abstract base classes `ProcessingUnit` and `PeripheralDevice`. With the green-colored `SimulationUARTAccess` and `SimulationI2CAccess`, two classes derived from the `RegisterAccess` and `CharacterAccess` interfaces as part of the policy-oriented HAL have been developed. They serve as the abstraction layer for communication interfaces to realize hardware-software interactions. The two red-colored UML classes in Figure 7.8 have been added to the system model for a data exchange based on the SDXP (cf. Section 6.2, p. 145 ff.). The `ModelLogger` class provides a system-wide logging interface used by hardware component models to generate simulation logs for the IPA and DPA. Additionally, both classes are utilized by the developed policy-oriented HAL to realize the data exchange. Note that `ModelConnector` and `ModelLogger` are external files represented by the stereotype «`File`». These external files are only required for the simulation and are not part of the UML model and the code generation process.

To model the behavior of the `UserApplication` UML class and, thus, the general behavior and the program flow of the software application model, the UML state machine shown in Figure 7.9 has been defined. It consists of the five states `InitSystem`, `Measure`, `Process`, `Send`, and `Sleep` as distinct execution phases of the software application model and a set of transitions for switching between these phases. Additionally, a set of events has been defined to trigger transitions, namely `evMeasure`, `evProcess`, `evSend`, `evSleep`, and `evActive`. They are generated within operations of the `UserApplication` class, such as `measure()` or `processData()` in Figure 7.8 (p. 183).

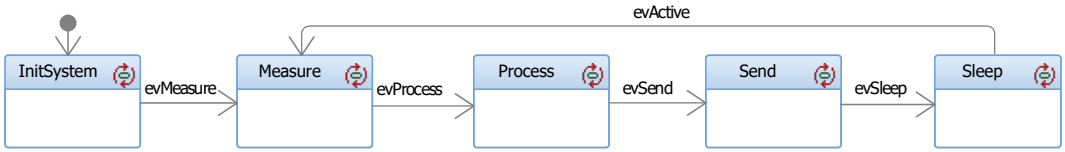


Figure 7.9: UML behavioral state machine of the software application modeled with IBM Rhapsody. Inner details are hidden. (UML 2.5 state machine diagram notation)

When executed, the software application model enters the `InitSystem` state, where the peripheral device class instances are initiated, the BME280 sensor and the RAK811 LoRa module are configured, and a `join` request to the TTN is initiated. After the TTN has been joined, the event `evMessure` is generated, resulting in a state change from `InitSystem` to `Measure`. During the execution of the `Measure` state, a single measurement with the Bosch BME280 sensor is performed (forced mode [53]), and measurement data are obtained. Afterward, the `evProcess` event is generated, forcing the software application model to enter the next state, where the acquired sensor values are analyzed and accumulated into a message. The message is then transmitted to a TTN gateway during the `Send` state. After the transmission has been completed, the software application model switches to the `Sleep` state in which the system is set into a low power mode for ten minutes. During that time, the RAK811 LoRa module and the Espressif ESP32 are put into their previously defined sleep

modes. When an `evActive` event is generated, the system returns to the `Measure` state, and the process of measuring, data processing, and data transmission is repeated. The functional behavior of UML operations and UML states defining the application logic are modeled in textual form as opaque behavior using C++, e.g., to facilitate the integration of existing device drivers, configure hardware component models, and generate events to initiate transitions. The illustration of opaque behavior has been omitted in Figure 7.8 and Figure 7.9.

Two energy-aware design patterns were applied during the development of the software application model. First, the software application configures the connected devices with their maximum possible speeds and transfer modes in the `InitSystem` state, which corresponds to the Race-to-Sleep design pattern (cf. Section 4.4.6, p. 111 ff.). This allows the beehive microclimate sensor node to switch back to a low-power mode more quickly. Second, the EAS design pattern (cf. Section 4.4.1, p. 95 ff.) has been applied to the Bosch BME280 sensor, where the oversampling for all three sensors (temperature, humidity, and pressure) has been reduced to one instead of 16. By this, the energy footprint of the sensor is reduced.

This section demonstrated the development of a system model by integrating hardware component models into a software model. The concept of the policy-oriented HAL (cf. Section 6.3, p. 149 ff.) has been successfully implemented for the I^2C and UART communication interfaces. During simulation, the hardware component models of the BME280 and RAK811 use these implementations to achieve low-level communication for sensor data acquisition and transmission. With the behavior modeled as a UML state machine, both the hardware and software layers of the IoT sensor node case study have been successfully modeled in UML to enable power consumption estimation and power fault detection in early development phases.

7.2.4 Scenario Definition

For the evaluation, test lab scenario S_{tl} has been defined, which is further divided into the three different sub-scenarios $S_{tl} = [S_a, S_t, S_s]$, defined as:

- S_a: Scenario for the active phase to cover measuring and data processing activities.
- S_t: Scenario for the transmit phase, where data is sent to a cloud application.
- S_s: Scenario describing the sleeping phase of the IoT sensor node.

Even while the configuration parameter, e.g., for the Bosch BME280 and RAK811, and the characteristics of the laboratory environment are considered static and remain unchanged during simulation, the advantage of defining sub-scenarios lies in the possibility to define and assign different energy-related NFRs for each relevant segment of the software application life cycle. For instance, S_{tl} may contain a requirement for average temperature values obtained by the Bosch BME280 in laboratory environments, which are expected to vary between 18-20 °C. Since the evaluation focuses on NFRs, the exact measured temperature is not relevant as long as it remains within limits defined by the scenario S_{tl}. In sub-scenario S_t, only the RAK811 is allowed to change its state to transmit a message to the TTN. For sub-scenario S_s, all components must be in a pre-selected low-power state (cf. Table 7.1. p. 178), where the MCU is assumed to operate in `DEEP_SLEEP` state and the RAK811 in the `Sleep` state, while the BME280 remains in the `Idle` state. Exemplary for the sub-scenario S_s, the following two power-related NFRs may be specified for the RAK811 LoRa module:

NFR1: The total energy of the LoRa module (`rak811`) during a single sleep phase period of 10 min shall not exceed $13.07 \cdot 10^{-3}$ J.

NFR2: The LoRa module (`rak811`) shall not consume more than 6.6 µA when the device operates in a low-power mode.

As a threshold for the evaluation in this thesis, the power-related NFRs may be fulfilled if the measured power consumption does not exceed 20 % of the expected power consumption specified in Table 7.1 (p. 178). Considering a sleeping phase of 10 min and an supply voltage of 3.3 V, an energy bug threshold can be derived from NFR1 and NFR2 and expressed as the tuple $\langle(13.07, mJ), (6.6, \mu A)\rangle$. If the measured power consumption stays below this definition, the system may be considered energy bug free for scenario S_s.

Measurement data is transmitted to the TTN along with additional metadata using a public gateway of the TTN located at a distance of 2.94 km from the *Model-Testbed*. The test lab scenario S_{tl} defines the position of the *Model-Testbed* as static so that the distance between the *Model-Testbed* and the TTN gateway does not vary during test case execution, resulting in a fixed configuration of the RAK811 LoRa module, e.g., with a spreading factor of 12 (cf. Table 7.2, p. 181). Furthermore, the scenario S_{tl} defines that the join process for the TTN requires only a single attempt. Due to the findings described in Section 7.2.2 (p. 178 ff.), the maximum duration of the first RX windows is considered to be 1300 ms, while the second RX window is omitted.

This section demonstrated the definition of scenarios (cf. Section 3.2, p. 81 ff.) to describe and specify properties of the environment that can directly influence functional and non-functional behavior. As a basis for evaluating the IoT sensor node, scenarios include constraints on the case study, which go beyond hardware and software aspects of the SUT. However, they are necessary for an evaluation based on NFRs. Considering the definition of energy bugs (cf. Definition 3.2, p. 85), limits and boundaries within a system can be defined in a machine-readable format to ease the process of evaluating energy-related behavior.

7.2.5 Power Consumption Estimation

This section demonstrates the capabilities of the presented power consumption estimation approach based on the beehive microclimate sensor node introduced in previous Sections 7.2.2 to 7.2.4. Additionally, this section compares the accuracy of the power analysis methods.

The simulation of the software application model was executed within the defined scenario S_{tl} for a total of 65 minutes, in which the UC^{2}E tool continuously obtained measurements, which resulted in a total of 219,148 measurement points. As an offline analysis, the diagram presented in Figure 7.10 has been automatically generated by the UC^{2}E tool after the simulation. It shows an energy trace based on DPA for an excerpt of 13 seconds in which the software application model completed an entire active cycle. The upper part of Figure 7.10 shows the total electric current consumption of all system components as a comparison between the expected (blue-colored line) and the measured power consumption (orange-colored line) directly from the *Model-Testbed*. The two plots in the middle and lower part describe the functional and power-related behavior of the ESP32 and RAK811 as blue and green lines.

If IPA is used instead of DPA, the energy trace would only contain a single line with the expected current consumption (blue-colored line in Figure 7.10). As a superset of IPA, DPA is able to trace both (cf. Section 5.4.2, p. 136 ff.). Since the expected current consumption is equal for both methods, the energy trace in Figure 7.10 can also be used to directly compare IPA and DPA.

Based on the simulation data, the UC^{2}E tool predicts a total energy consumption of 14.80 J for the complete test case execution of 65 minutes, e.g., as illustrated in Figure 7.13 for the

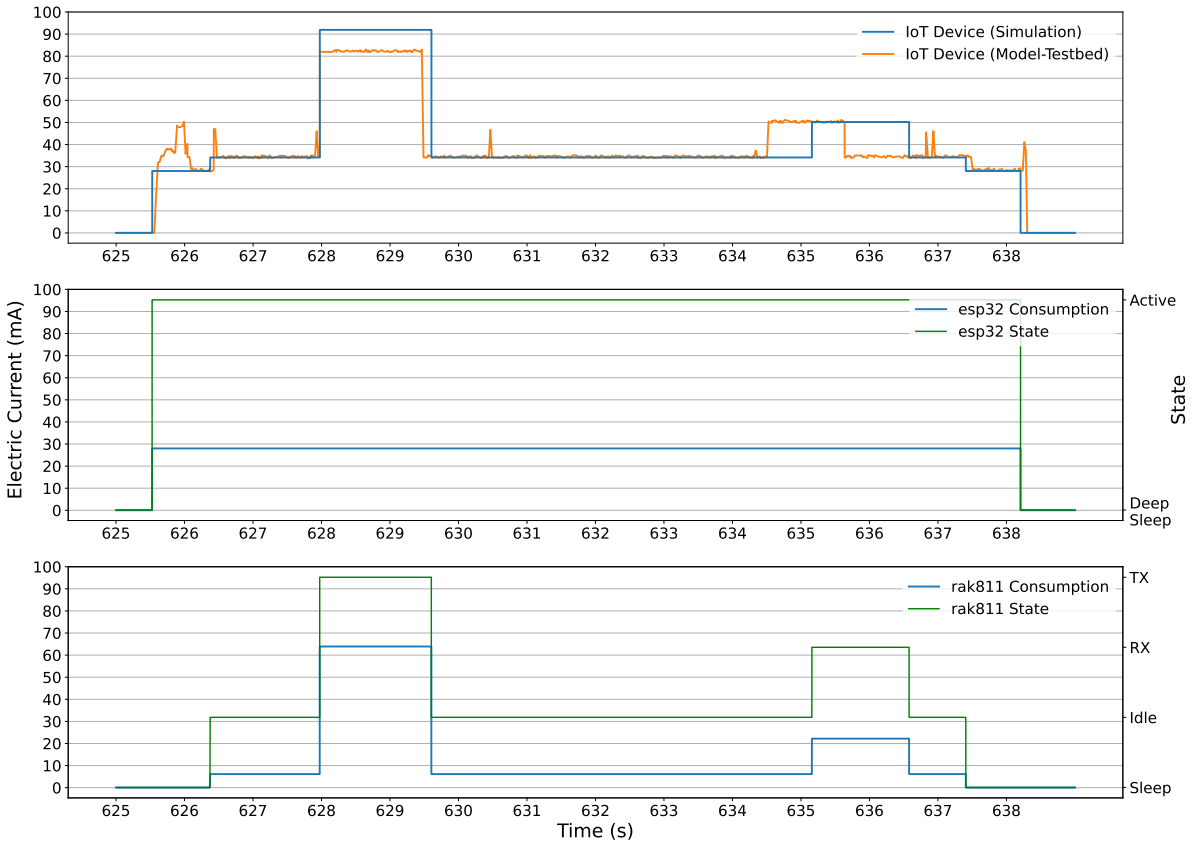


Figure 7.10: Energy trace generated by the UC^{2}E tool, adapted from [341]. The upper part shows the total electric current consumption of the system as a comparison between IPA (blue-colored line) and DPA with *Model-Testbed* execution (orange-colored line). Diagrams in the middle and lower parts show the expected functional (green) and power-related (blue) behavior of the ESP32 and RAK811. The results of the BME280 have been omitted due to their negligible impact.

detection of energy bugs in Section 7.2.6 (p. 189 ff.). In contrast, an energy consumption of 14.05 J has been physically measured by the Otii Arc connected to the *Model-Testbed*. For the presented case study, the evaluation revealed a total error of 5.38 % between expected (IPA) and measured power consumption (DPA).

The total power consumption of the SUT during the active phase, as exemplary shown in the upper part of Figure 7.10, can primarily be assigned to the ESP32 MCU and the RAK811 LoRa module. While the MCU remains in the active state (green line) for the simulation time between 625.5 s and 638.2 s with a static (from a macroscopic point of view) power consumption of 28 mA (blue line), the logged behavior of the RAK811 hardware component model indicates a more dynamic behavior. For the time between 625 s and 626.2 s, the RAK811 operates in `Sleep` mode with a power consumption of 5.5 µA. At 626.2 s, the software application model enables the RAK811 to initiate a data transfer. The RAK811 switches to the `Idle` mode, and the current consumption increases to 6.1 mA. The transmission starts at 628 s and is finished at 800 ms later. For that time, the hardware component model of the RAK811 expects the hardware component to operate in `TX` mode with a current consumption of 63.9 mA. The RX

window is opened automatically by the RAK811 module, which the corresponding hardware component model predicts for the period from 635.1 s to 636.7 s with a current consumption of 22.2 mA. Afterward, the RAK811 returns to the `Idle` mode and is put back to the `Sleep` mode by the software application mode at 637.4 s.

However, the excerpt of the energy trace in Figure 7.10 contains some deviations between the estimated and measured power consumption, which are highlighted with the red-circled letters (A) to (D) in Figure 7.11. Part (A) in Figure 7.11 shows the three-step startup phase of the MCU, executing the first and second stage bootloader followed by the application startup phase [111]. The three phases can be broadly assigned to the segments of the current consumption, which are described as (A) in Figure 7.11. However, it is unclear which specific sub-components of the MCU are responsible for the current consumption change, so detailed modeling is impossible. In addition, evaluations revealed that optimizations at the bootloader level result in reduced execution times. As the evaluation of the simulation log indicates, the energy model of the ESP32 hardware component model does not contain a fine-grained representation of the startup phase (cf. center of Figure 7.10). However, by refining the energy model, this phase may be considered by determining an average value for the duration of the startup phase and using the PAP to annotate all transitions from the initial and low-power states to the active state.

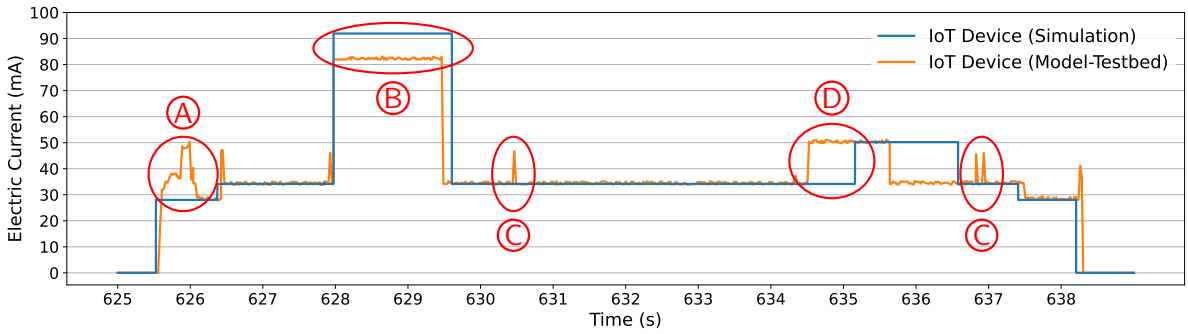


Figure 7.11: Energy trace for the case study evaluation with simulation deviations highlighted.

Part (B) in Figure 7.11 shows the transmission phase of the RAK811, where the model predicts a higher current consumption for a slightly longer period than actually measured. As shown in Table 7.1 (p. 178), the power consumption has been measured prior to the test execution and differs significantly from the data provided in the datasheet of the sensor [351]. For this specific test setup and defined scenario, the drift may result from inaccuracies of the values defined by the PAP in this state, e.g., due to a measurement error in the previously performed measurement of this thesis and environmental effects, which may lead to parameter changes during testing [73, 78, 287]. However, the current consumption value previously obtained still showed the smallest deviation compared to the values provided by the datasheet [317]. The length of the TX windows is determined by the modeled expression (cf. Listing 7.1, p. 181) based on calculations provided in the datasheet [351]. Again, if conditions during the test case execution change, the duration of the `TX` state may be affected. Another reason is time delays of DPA, which are discussed in detail in Section 7.3 (p. 191). In a further step to refine the hardware component model for future simulations, values and expressions may be adjusted in the corresponding stereotypes of the PAP.

The artifacts (C) in Figure 7.11 are effects introduced by the DPA method and result from the instrumentation of the *Model-Testbed* using the UART communication interface, since every low-level interaction of a hardware component model requires communication with the *Model-Testbed*. The analysis of the simulation logs has shown that such artifacts marked as (C) can be observed during the transmission of each *Model-RPC* message, resulting in additional current consumption caused by the UART interface during this time.

A time-related drift in the simulation is marked as (D), which shows the start of the first RX window by the RAK811 module. This particular example of extensive delay arises during the simulation process and is further amplified due to the high load of the host system. More precisely, since the energy model (cf. Figure 7.6, p. 180) is modeled in such a way that it automatically switches to the `RX` state after an internal timer of 5000 ms has elapsed and triggers the `evIdleRX` event, deviations can be traced back to the simulation. The effects of DPA and time delays are discussed more in-depth in the following Section 7.3 (p. 191 ff.).

As a contribution to answer RQ4, the evaluation in this section has demonstrated the applicability of the power estimation approach using a realistic case study of an IoT sensor node in early phases of the MDD process as described by the V-model in Figure 2.21 (p. 59). The estimation process has been performed as a MiL test (IPA) and a HiL-like test (DPA). The loss of accuracy and the absence of environmental effects are compensated by the advantages of IPA, e.g., real-time simulations and design space explorations without physical hardware. It has been shown that DPA enables simulated software application models in IBM Rhapsody to communicate directly with real hardware components by using a *Model-Testbed* in early development phases while skipping cost and time intensive steps, e.g., *edit-cross-compile-flash-debug* cycles [30, 393]. Since the power consumption estimation relies on protocols such as SDXP, IPA and DPA are not limited to the simulation of UML classes and UML state machines as used in this section. Instead, one may also apply the power consumption estimation approach on partial models or even single UML activity diagrams to evaluate parts of the software application that would not be executable on a physical embedded system without major effort. During the evaluation, the UC^{2}E tool was able to trace the states of each hardware component, which not only creates measurement transparency but also allows developers to detect any misbehavior of individual components. Since the concept of *Model-Testbeds* is able to handle low-power modes of the underlying MCU, only the active state of the MCU may differ due to the fact that the software application is not executed directly on the embedded system. Considering the provided concept of a policy-oriented HAL combined with communication protocols such as *Model-RPC*, the power-related behavior of peripheral devices is not expected to be significantly different when using DPA.

7.2.6 Detection of Energy Bugs

This section describes the abilities of DPA to detect and identify energy bugs discussed in Section 3.3 (p. 84 ff.). Referring to the test lab scenario S_{tl} defined in Section 7.2.4 (p. 185 f.) as part of the case study, a set of two power-related requirements, NFR1 and NFR2, have been defined for the sleeping phase (sub-scenario S_s) of the SUT.

These NFRs are considered while evaluating the beehive microclimate IoT sensor node as a proof-of-concept for an energy bug detection. Figure 7.12 shows an extraction of the energy trace for the case study evaluation presented in Section 7.2.5 (p. 186), in which the components of the *Model-Testbed* switch from an active to a low-power mode. For sub-scenario S_s, the power consumption of the system shown on the left side in Figure 7.12 and labeled

as (A) is higher than expected. Based on the energy trace generated by the UC^{2}E tool within DPA, the RAK811 LoRa module operates in the `Idle` state (B), while the ESP32 MCU is already set into the `DEEP_SLEEP` mode. Since both NFRs have been violated, a presence of an energy bug is very probable. Software developers may use the information provided by the UC^{2}E tool to further isolate the source of the energy bug. In the provided example, the source turned out to be a software-related energy bug (Type *C*) due to a missing operation call to initiate a state change of the RAK811 from `Idle` into the `Sleep` state. The right side of Figure 7.12 shows the execution of the revised software application model.

The overall impact of the energy bug is illustrated in Figure 7.13. The accumulated power consumption for the beehive microclimate sensor node with a faulty software application model is illustrated as blue lines. The accumulated power consumption is shown as red-colored lines for the revised software application model. A dashed line shows the estimation based on IPA, while straight lines are measurements with DPA.

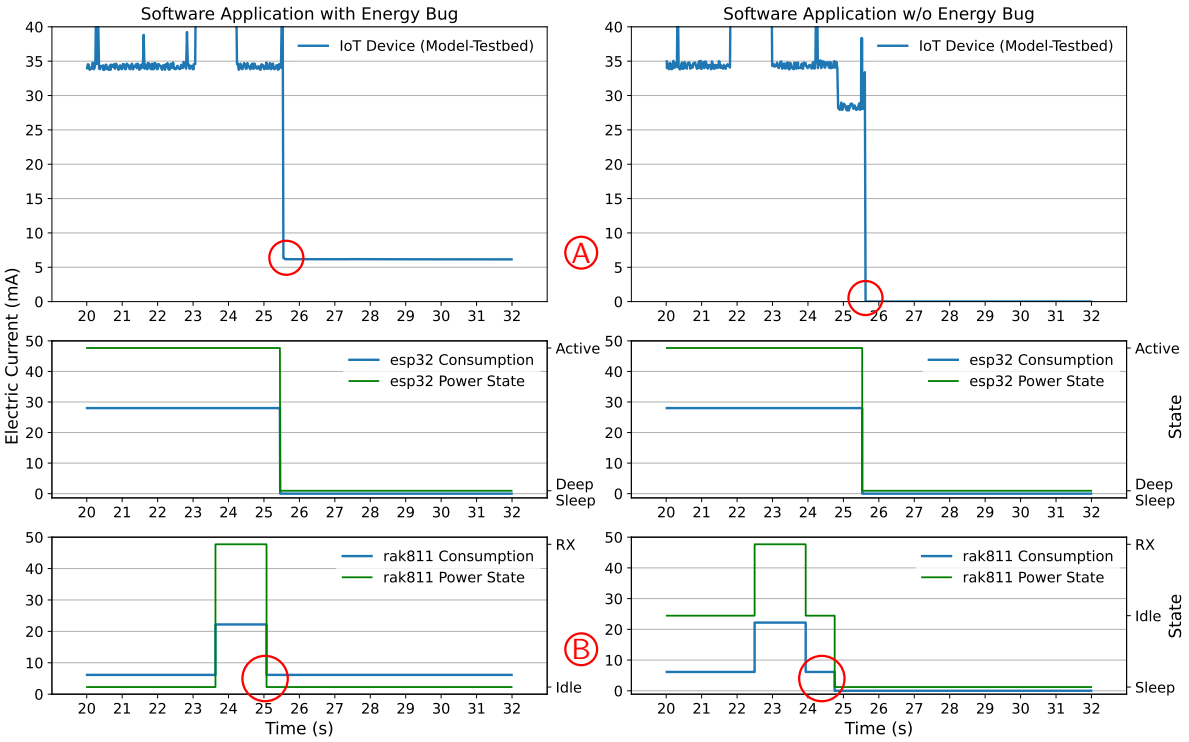


Figure 7.12: Analysis of a software application with (left) and w/o (right) a software-related energy bug. Technical indicators for the presence of an energy bug are highlighted in red. The overall consumption (A) is higher than expected. The RAK811 LoRa module stays in the `Idle` state (B) while the ESP32 MCU is set into the `DEEP_SLEEP` state. Adapted from [341].

During the evaluation of the case study, another unexpected behavior was coincidentally observed, which turned out to be a Type *A* energy bug. A sporadic and undesired behavior caused the RAK811 LoRa module to enter the `Idle` state while the SUT was still in the sleeping phase. Further analysis revealed an open circuit due to faulty wiring of the UART interface between the MCU and the LoRa module of the *Model-Testbed*. The RAK811 sporadically interpreted the undefined behavior of the GPIO as a wake-up signal forcing it to switch back to

the `Idle` state. Since the behavior contradicts both NFRs, it was classified as an energy bug. In general, the UC^{2}E tool is able to detect such hardware-related energy bugs by recognizing a sudden increase in the measured current consumption value without a prior state change of a hardware component model. Since the difference between the measured current consumption value before and after the increase is approximately equal to the corresponding electric current value of a state or transition, the UC^{2}E tool is able to identify the hardware component model instance causing the misbehavior. The hardware-related energy bug has been resolved by adding pull-up resistors to the UART TX and RX lines.

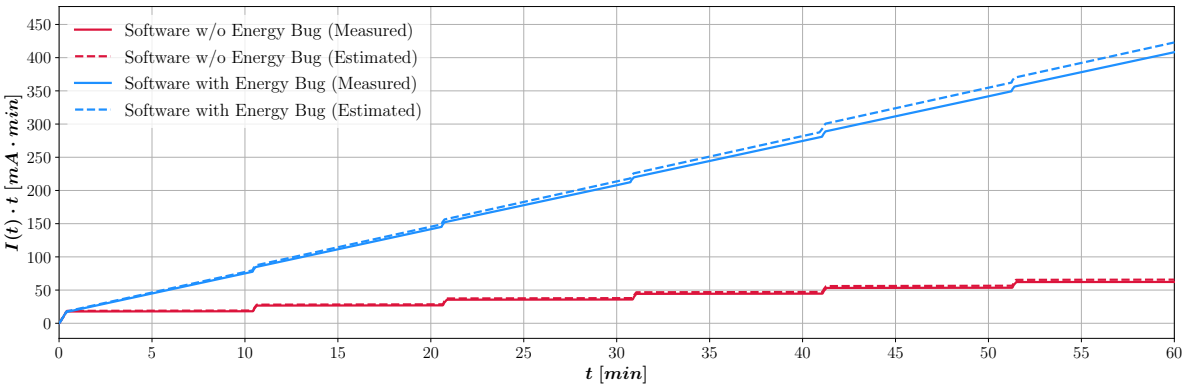


Figure 7.13: Comparison of the beehive microclimate software model energy consumption with and without energy bug forcing the RAK811 to remain in a higher power mode.

This section has shown the capabilities of the approach to detect energy bugs. Based on power-related NFRs specified as part of scenarios in Section 7.2.4 (p. 185 ff.), the detection of software-related and hardware-related energy bugs has been demonstrated. Furthermore, all concepts presented in Chapter 3 (p. 77 ff.) and Chapter 5 (p. 115 ff.) could be integrated into a single use case and are successfully applied in a proof-of-concept.

7.3 Overall Performance of DPA

This section evaluates the efficiency of the proposed DPA method by using basic examples to determine performance metrics such as delays, tradeoffs, and limitations. The evaluation of DPA includes the analysis of time delays in Section 7.3.1 and power-related overhead discussed in Section 7.3.2. For statistical analysis, a benchmark to analyze the time delay and power-related overhead has been defined.

7.3.1 Investigation of Time Delays

In general, time delays of the DPA approach may lead to less accurate results caused by additional communication delays between the MDD tool, UC^{2}E tool, and the *Model-Testbed*. This delay may, for instance, cause hardware components to remain in a high-energy state much longer than intended. Compared to a native execution of the software application on the hardware platform, it is likely that the presented approach introduces time delays. Depending on how significant these delays become, the derived energy trace may be less beneficial for developers. Therefore, evaluating the time delay caused by DPA is crucial to determine the

effectiveness of the proposed power estimation approach. A basic software application that periodically toggles a single GPIO between high and low states at a fixed interval of 1000 ms is used to evaluate time delays. For the measurement with Otii Arc, an LED is connected to the GPIO as a consumer to detect a change in the current consumption. This example has the smallest *Model-RPC* message size, and switching a GPIO is considered the most basic function of the MCU. Therefore, the resulting delay defines the lower limit for any action performed on the *Model-Testbed* when using DPA.

In the following, the evaluation is divided into two parts. The first part analyzes the delays caused by the communication between the UC^{2}E tool and the *Model-Testbed*. The second part deals with the delays of the used MDD tool IBM Rhapsody.

Delays due the communication between the UC^{2}E tool and the Model-Testbed

The first analysis focuses on delays between the UC^{2}E tool and *Model-Testbeds*. The basic example modeled with IBM Rhapsody is compared with a native application with the same functionality flashed and executed on the ESP32-based *Model-Testbed*. The same hardware was used for both measurements, flashed with either the developed firmware (cf. Section 6.5.2, p. 163 ff.) or the base software application using FreeRTOS [13]. Figure 7.14 shows a single switching event for the GPIO from a low to a high state as a comparison between DPA (green-colored line) and the native execution of the software application (red-colored line). The power

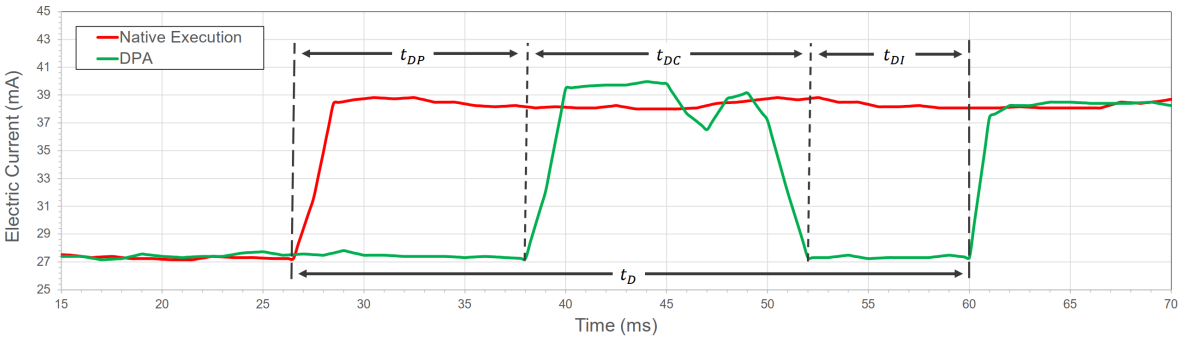


Figure 7.14: Time delay for activating a single GPIO as a comparison between the software application executed natively on the MCU under FreeRTOS (red-colored line) and the DPA approach (green-colored line), published in [341].

consumption of the *Model-Testbed* in milliamperes is shown on the y-axis, while the x-axis describes the execution time in milliseconds. For synchronizing both measurement series in time, the UC^{2}E tool generates a time-referenced marker for each incoming message of the MDD tool. The marker and measurement data of the Otii Arc are stored in a single log file and analyzed after the benchmark has been completed. For the *Model-Testbeds* with GPIO in a low state (LED off), the current consumption of about 27.5 mA corresponds to the current consumption of the ESP32 MCU (see Table 7.1, p. 178). With GPIO in high state (LED on), the current consumption increases by 11 mA to about 38.5 mA. As an indicator that the GPIO has been switched and the LED is enabled, the first measured value with an increase in current consumption for both use cases was selected. Based on the marker and identified measured value, a deviation between both use cases could be identified, which is referred to as delay t_D. Let the total time delay t_D between the native implementation and the DPA

approach be defined as:

$$t_D = t_{DP} + t_{DC} + t_{DI} \tag{7.1}$$

where elements defining t_D can be considered as distinct steps in the DPA approach. The processing delay t_{DP} refers to an additional offset caused by the message processing of the UC^{2}E tool. It defines the period between the arrival of a message and the completion of the message processing, including the parsing and creation of SDXP and *Model-RPC* messages. This delay mainly depends on the underlying host system, e.g., the load and scheduler of the operating system executing the UC^{2}E tool and IBM Rhapsody. In Figure 7.14, the delay t_{DP} has a length of 11 ms.

The communication delay t_{DC} defines the time-related latency resulting from the communication between the UC^{2}E tool and the *Model-Testbed*. In general, the delay t_{DC} scales with the size of the *Model-RPC* message and the transmission speed and is expected to be static for the same type of messages and configured transmission speed. Since the *Model-RPC* GPIO message requires the smallest number of bytes, the t_D in Figure 7.14 may be considered as the minimum delay for *Model-RPC* messages transmitted with the configured baud rate of 115,200 bps. For example, *Model-RPC* UART messages (cf. Section 6.5.4, p. 168 ff.) require more bytes since they contain a buffer with the data to be transmitted as strings. The selected baud rate of 115,200 bps for the CP2102 USB-to-UART bridge represents the default value of the ESP32 and may differ for other MCUs. The MCU also supports baud rates up to 1,500,000 bps which would significantly reduce the time-related overhead defined by t_{DC} [112].

The message interpretation delay t_{DI} is related to the time between the reception of a message and switching the GPIO, where the most significant portion of the delay t_{DI} is required for parsing and processing the string-based *Model-RPC* message on the *Model-Testbed* to switch the logical state of the GPIO. In Figure 7.14, t_{DI} has a length of 8 ms. Note that based on the capabilities of the MCU, the t_{DI} may vary and is *Model-Testbed* specific.

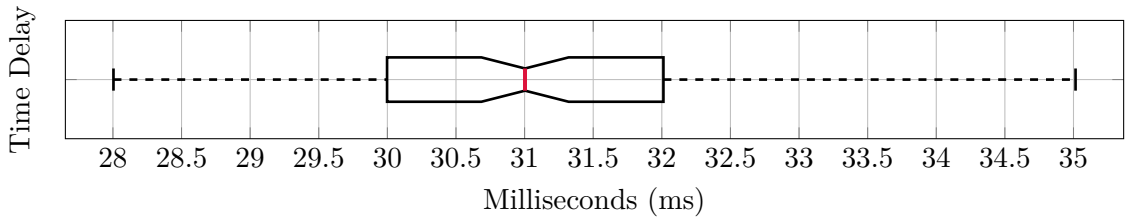


Figure 7.15: Statistical evaluation of the time delay t_D for switching the logical state of GPIO in DPA based on a benchmark with 100 iterations as a box plot (published in [341]).

Figure 7.15 shows the analysis of the total time delay t_D for the presented approach. The values of t_D have a maximum value of 35.0 ms and a minimum value of 28.0 ms. All measured delays fit within limits without detecting outliers during the execution of the benchmark. However, the median $\tilde{x}$ of t_D is 31.0 ms, the lower quartile $Q_1 = 30.0$ ms, and the upper quartile $Q_3 = 32.0$ ms. Furthermore, an average value of 30.9 ms with a standard deviation $\sigma = 1.5$ has been determined. The results show that the communication between the tool and the testbed is subject to only minor fluctuations, which are also influenced by factors not directly related to the approach, such as the host system. The minimum delay between the generation of low-level hardware access and the actual execution is expected to be 30.9 ms

on average. This delay becomes particularly important when software-hardware interactions are time-critical and require execution times below the measured delay, e.g., when a series of commands must be sent to a peripheral device. However, there still exists sufficient potential for optimization, for instance, by increasing the transfer rate, reducing the message size, or optimizing the message format.

Delays introduced by IBM Rhapsody

The second analysis focuses on the simulation environment of IBM Rhapsody to obtain a complete impression of the overall DPA approach. For this, two additional benchmarks have been performed to analyze the simulation accuracy of the software model and to quantify the data transmission delay of the local TCP socket communication between the IBM Rhapsody and the UC^{2}E tool. The results of both benchmarks are illustrated as box plots in Figure 7.16.

As mentioned at the beginning of this section, this benchmark has been designed as a periodic test where the simulated model triggers a GPIO at a fixed interval of 1000 ms. To analyze possible delays, each message generated by the software model has been recorded, and the unique timestamp extracted. The time deviation Δt for two consecutive messages m_n and m_{n+1} generated by the software application model at simulation time t_n and t_{n+1} can be determined by calculating $\Delta t = (|t_n - t_{n+1}|) - 1000$. The time deviation of the simulation is illustrated as a boxplot in the upper part of Figure 7.16. A value of 0 ms for Δt means that two SDXP *Action* messages have been generated without any additional delay of IBM Rhapsody or the host system, respecting the expected interval of 1000 ms. Note that no negative deviations were recorded during the benchmark. The results show an average of $\Delta t = 7.0$ ms, a median of $\tilde{x} = 5.0$ ms, and a deviation $\sigma = 6.3$ ms. The lower quartile Q_1 and the upper quartile Q_3 are located at 0 ms and 20.0 ms, respectively, while the highest delay measured was 27.0 ms. Compared to the expected interval of 1000 ms for switching the GPIO, the messages are generated on average every 1007 ms, but at most after 1027 ms due to the simulation environment and host system (e.g., load, operating system, and scheduler).

The lower part of Figure 7.16 shows the transmission delay for the generated message between the IBM Rhapsody MDD tool and the UC^{2}E tool. The SDXP *Action* messages (cf. Section 6.2.1, p. 145 ff.) were received by the UC^{2}E tool after an average time of 31.6 ms with

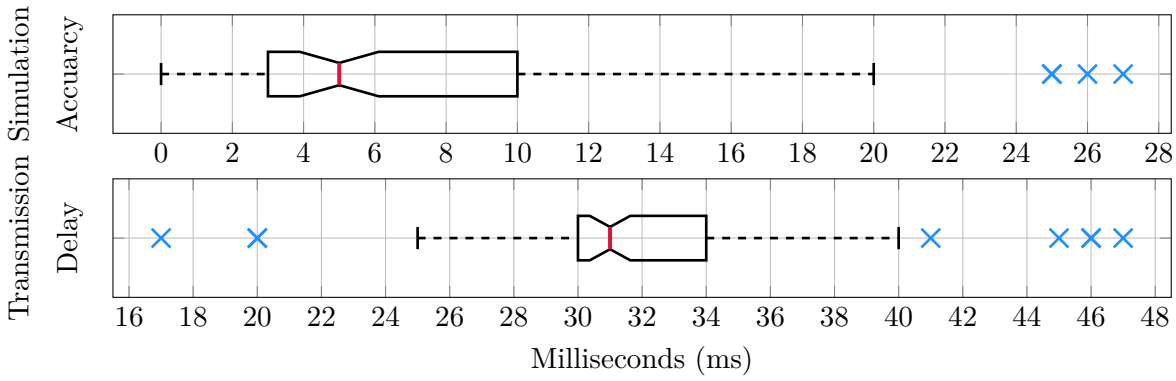


Figure 7.16: Time delays of the simulation environment and the message transmission during a benchmark with 100 iterations as box plots (published in [341]).

$\tilde{x} = 31.0$ ms. Without including outliers, the lower quartile Q_1 is located at 25.0 ms and the upper quartile Q_3 is 40.0 ms. It should be noted that the environment mainly causes both sources of time delays shown in Figure 7.16, e.g., due to the task scheduler and the current load of the host's operating system. Considering a minimum value of 17.0 ms and a maximum value of 47.0 ms, the range of 30.0 ms is a strong indicator that the host system used for DPA might not be powerful enough to handle time-critical and low-latency hardware-software interactions, e.g., software-based generation of PWM signals (Soft-PWM) which require a constant timing. This is supported by the fact that the high load of the host system due to the simultaneous execution of DPA and additional applications, e.g., for further documentation, led to data loss of the Otii desktop application in some test runs and gaps in the data recordings and diagrams. In general, the delays mentioned above may have a negative impact on the overall power estimation process. This is especially true for inaccuracies within the simulation environment, as this distorts the system behavior. The transmission delay causes a right shift of the measurement curve and does not affect the power consumption estimation approach if and only if this delay can be considered as a static offset.

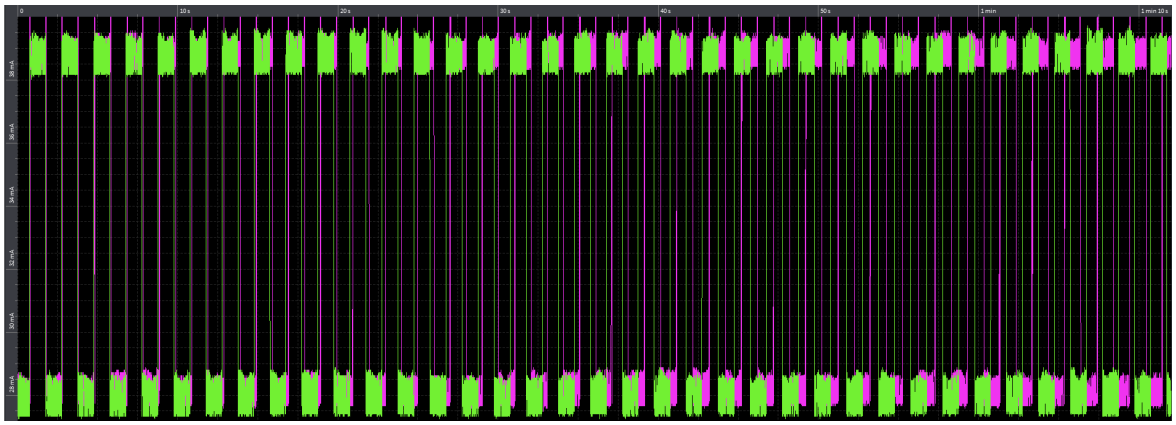


Figure 7.17: Effects of time delays for the example application with a one second interval. The graph shows native execution (green line) and the simulation using DPA (pink line).

To illustrate the effect of time delays, Figure 7.17 shows a plot of the current consumption generated by the Qoitech Otii desktop application as the direct comparison between the native execution of the software application switching an LED as a green curve and the simulation using DPA as a pink curve. Both graphs were synchronized in the Qoitech Otii desktop application for better visualization by the first rising edge. The continuous drift through time delays analyzed in this section causes the simulated software application within DPA to lag for almost one period after a simulation time of 70 s. Note that the impact of time delays becomes more significant as the number of actions within a period increases.

This section provided an in-depth analysis of the overall performance and characteristics of DPA. A minimal delay of 30 ms for direct interactions between the UC^{2}E tool and the *Model-Testbed* defines the lowest limit for timing requirements of the systems to be evaluated with DPA. Additional delays not related to the presented approach are introduced by the operating system executed on the host system, e.g., due to the scheduler and the general load affecting parts of the simulation, such as timers. In summary, the delays of the approach can be considered acceptable for a power consumption estimation in early MDD phases.

7.3.2 Power and Timing Tradeoffs

As discussed in Section 7.3.1, several tradeoffs for the presented DPA method exist. While *edit-cross-compile-flash-debug* cycles [30, 393] can be avoided for a faster and earlier evaluation, other power and timing tradeoffs affect the accuracy of DPA. Such tradeoffs include:

- *Overhead* due to the implementation of monitoring functionalities and additional communication with the *Model-Testbed* to perform hardware-software interactions.
- *Inaccuracies* of the estimation due to unconsidered hardware components.

Generally, tradeoffs have a *subtractive* or *additive* character whereby the terms refer to additional operational steps in the post-processing of energy traces to adjust and improve the overall estimation.

The term *subtractive* describes tradeoffs introduced by the DPA that must be removed from energy traces for a highly accurate power consumption estimation. These tradeoffs are introduced due to the monitoring and instrumentation of the *Model-Testbed*, which is used as external instrumentation to perform hardware-software interactions. As shown in Figure 7.11 (p. 188), sending instructions to the *Model-Testbed* results in additional power consumption for the communication between the UC^{2}E tool and the *Model-Testbed* via UART. Since this additional power consumption does not exist when a software application runs natively on an embedded system, this power-related overhead must be subtracted from an energy trace to obtain a more accurate estimation. This is also true for time-related overhead (cf. Section 7.3.1, p. 191 ff.). When referring to the evaluation of the beehive microclimate sensor node case study, the impact of this tradeoff can be considered marginal and extensive post-processing is negligible. However, the tradeoff may become more significant with increased communication.

The term *additive* describes additional power consumption caused by other hardware components and peripherals of the *Model-Testbed*, for which no proxy (hardware component model) exists. For the proposed power analysis methods, the additional power consumption of communication interfaces has to be added in post-processing, e.g., by counting low-level interface accesses and adding a fixed offset for each send and receive operation. With DPA, on the other hand, the additional electric current consumption is part of the generated energy trace but might not be associated with a specific hardware component model instance. Instead, the power consumption from unconsidered hardware components becomes part of the background noise. In this thesis, additive tradeoffs are mainly related to the communication between peripheral devices within the *Model-Testbed*, e.g., I^{2}C or UART. For instance, while the energy model of the RAK811 LoRa module reflects the characteristics of the hardware component itself, the additional overhead caused by the UART communication interface used between the ESP32 and the RAK811 is not considered. Concerning the current consumption of the overall system, the effect of such communication interfaces can be considered as being low [212, 252]. However, for a more accurate power analysis, the policy-oriented HAL might be extended with energy models for each communication interface.

The discussed tradeoffs mainly occur due to two factors. The first factor arises from disturbing the observed system, commonly known as the observer effect. The second factor results from abstraction as part of the UML modeling process. Unconsidered hardware components may not falsify the measurement but may negatively affect the estimation and evaluation of NFRs if their impact becomes too large. Despite the limitations described in this section, the results provided in this chapter indicate that further scientific investigations are reasonable.

Chapter 8

Conclusion

This thesis provides a novel and innovative approach to model and estimate the power consumption of software applications in MDD for embedded systems. In the following, the contributions and findings are summarized. This chapter concludes with an outlook for future work based on the results of this thesis to be discussed. This chapter concludes with an outlook on open topics and future work based on the results of this thesis.

8.1 Summary

In areas such as IoT, embedded systems are expected to be resource-constrained in terms of performance, power, timing and costs. Especially when battery-powered embedded systems are placed in harsh environments or buried underground [146, 409], the supply of power is a major challenge. In many cases, replacing or recharging the power source is either impossible, impractical, or results in higher costs. Besides battery capacity, managing the energy an embedded system consumes is one of the most critical factors in determining how long the system will last.

Developers and engineers have to deal with the increasing complexity of embedded system designs due to the variety of processor architectures, communication interfaces, and a growing number of devices with distinct functionalities, such as sensors and actuators. While power consumption is commonly associated with the hardware layer, little attention has been given to the software layer, which controls and directs most hardware activities. Energy awareness is often neglected in the embedded software development process, and developers and engineers may be unaware of how to reduce the energy impact of software applications [289, 301]. For the software engineering process, approaches such as MDD may be used to manage the complexity of software applications for embedded systems. This enables analysis at the design and architecture level. However, most approaches and tools in MDD are focused on functional and timing aspects of software applications, while they lack support for an early analysis of NFRs. As a result, such requirements are evaluated at the end of the development cycle, where errors and misbehavior can cause additional time delays and costs.

This thesis presents a novel and innovative model-driven approach to estimate the power consumption of software applications for embedded systems in early design phases. The approach is realized by a set of contributions, summarized in the following according to the individual subjects of the four main *Research Questions (RQs)* elaborated in this thesis. As a comprehensive contribution, a developer workflow has been specified to combine all proposed

contributions of this thesis into a coherent process, starting from initial modeling to a power consumption estimation in early design phases using physical hardware.

Formal Definition of Energy-related Behavior and Defects (RQ1)

For an early analysis of software applications, a formal definition of energy-related behavior as a set of NFRs is required. The elaborated contributions related to this topic can be summarized as follows:

- Introduction of the two metrics energy quota E_{qu} and the maximum current demand I_{dmax} to describe power- and energy-related NFRs and to define the boundaries for an energy bug-free system.
- A novel classification of energy bugs independent of specific device types.
- The concept of scenarios as a set of conditions and constraints to specify aspects of the environment and the system when executing test cases.

The use of metrics for the formal description of power- and energy-related NFRs allows the precise specification of the expected behavior and the definition of boundaries for an energy bug-free system or subsystem. The violation of one or both metrics may indicate the presence of an energy bug. Since energy-related behavior strongly depends on the context and environment in which the hardware system and software application are deployed, the concept of scenarios has been introduced. They define a set of conditions and constraints, e.g., environmental conditions and system configurations, which apply for a specific amount of time during the execution of test cases. By this, environmental changes can be reflected within a test case for a more realistic evaluation of reactive systems.

With the provided specification of NFRs and the introduction of scenarios, the energy-related behavior of a system can be specified and evaluated.

Best Practices and Design of Energy-aware Software Applications (RQ2)

When addressing power and energy issues in software engineering, the availability of documented best practices is essential to implement the most appropriate solution. Such paradigms are referred to as software design patterns. However, current design pattern templates do not cover non-functional aspects such as power and energy consumption. To address this gap, the following contributions are provided:

- A novel framework and a design pattern template for identifying and uniformly describing energy-aware software design patterns.
- A design pattern catalog consisting of one novel and five redefined best practices for an energy-aware design of software applications.

As part of the framework, the proposed design pattern template provides new fields for the quantitative description of energy consumption and current demand as characteristics of an energy-aware software design pattern. With the *energy balance* EB_P and the *efficiency factors* η_P and η_C, the framework also introduced a set of new metrics as benchmark parameters for a quantitative evaluation of energy-aware software design patterns in terms of efficiency and effectiveness. EB_P defines a balance equation to indicate possible energy savings, whereas

η_P and η_C define effort-saving ratios to provide a quantitative evaluation and to describe the efficiency of possible energy savings resulting from applying an energy-aware design pattern. Each energy-aware design pattern description is also extended with a standardized graphical representation to outline the energy, temporal, and computational behavior aspects. Moreover, based on the uniform design pattern template, a catalog of energy-aware design patterns has been introduced in this thesis.

Researchers and software developers can use the introduced design pattern framework to uniformly document new design patterns based on the provided pattern template, metrics, and graphical representation. They can also select appropriate solutions from the provided design pattern catalog when modeling and evaluating energy-aware applications. Furthermore, the uniform representation achieved by the design pattern template may help to speed up the selection of best-fitting design patterns and the overall decision-making process.

The idea of the design pattern framework for energy-aware software design patterns has been noticed by the scientific community, where the related publication [337] was awarded with the *Best Student Paper Award* by the program committee reviewers of the conference. Additionally, the ideas and concepts of the publications [337, 392] have been included in the fourth edition of the book *Software Architecture in Practice* by Bass et al. (2021) [38] as a curriculum for future software engineers.

Joint Modeling of Functional Software Application Models and Energy Behavior (RQ3)

Since software applications control and drive most hardware activities, they have a significant impact on overall power consumption and define the runtime behavior of the embedded system. To model power-related aspects of the software application, hardware-software interactions and the dynamic behavior of hardware components must be considered. However, approaches in MDD lack the capabilities to combine those power-related aspects with the software application model. This thesis addresses this gap by providing the following contributions:

- A novel system-wide modeling approach based on hardware component models covering MCUs and connected peripheral devices.
- A UML-based description of hardware components that can be combined with a software application model to define a system model.
- The introduction of a UML profile to model energy-related aspects.

For extending software applications with power-related properties of the system, the concept of hardware component models has been presented as UML-based descriptions of hardware components, including an *energy model* to specify the power-related behavior over time. Hardware component models can be combined with a software application model to define a system model. Based on the interactions between the software application model and hardware component models, power-related aspects become visible.

For the specification of power-related aspects and NFPs in UML, the PAP UML profile has been developed. Based on the MARTE profile, the PAP introduces a set of new stereotypes and data types to define, e.g., the electric current consumption and execution time. Stereotypes of the PAP can be applied to states and transitions of UML state machines and attributes and operations of UML classes. The PAP has been designed to model dynamic power-related

aspects of hardware components when executed along with software application models. By this, the UML profile represents the central element for developing energy models.

Based on the hardware component model definition, a seamless integration of hardware properties into the software model is achieved. The resulting system model is used for the system-wide power estimation in early phases of energy-transparent development in MDD.

Early Evaluation of Energy-aware Software Applications in MDD (RQ4)

Early testing is one of the key factors in improving the quality of software applications and systems. It enables developers to identify and resolve issues in early development phases before the product reaches the integration phase. If no methods and analysis tools are available to support developers in evaluating software models early, the analysis process may be postponed and performed in later development stages. However, time and effort needed to address non-functional misbehavior in later phases is significantly higher. Since MDD tools for UML offer only a limited simulation environment and lack support for the evaluation of NFRs, an external evaluation approach was developed, leading to the following contributions:

- Two methods for estimating power consumption in early MDD phases with and without the use of a physical hardware platform.
- A case study of an IoT sensor node as a proof-of-concept to demonstrate the potential of the power consumption estimation approach.

With *Indirect Power Analysis (IPA)* and *Direct Power Analysis (DPA)*, this thesis provides two methods for a power consumption estimation in different development phases. IPA provides a simulation-based rapid power estimation without requiring a hardware execution platform. DPA is based on a novel in-the-loop approach that utilizes a *Model-Testbed* to enable direct interactions between the simulated software model and a physical embedded system. For this, *Model-RPC* has been introduced as a universal and lightweight communication protocol allowing hardware accesses of the simulated system model to be forwarded and executed on the *Model-Testbed*. By reproducing hardware-software interactions, *edit-cross-compile-flash-debug* cycles can be avoided to enable an evaluation in early MDD phases. For an accurate power consumption estimation, a measuring device connected to the *Model-Testbed* obtains measurements during testing. Another key advantage of DPA is the ability to obtain real data from peripheral devices and to use communication interfaces associated with IoT to enable real data transfer. Due to this, estimations of DPA can be considered very accurate as they are based on the actual behavior of peripheral devices. As a platform for rapid prototyping, the concept of *Model-Testbeds* follows a universal approach in which hardware components can be exchanged and the firmware dynamically configured. Three *Model-Testbeds* based on different MCU architectures have been developed to prove platform independence.

For the power consumption estimation of software models, the UC^{2}E tool has been introduced. As a central component of IPA and DPA, the UC^{2}E tool controls the communication and interaction between the MDD tool, measuring devices, and the *Model-Testbed* during the simulation. For direct data exchange between the simulated system model and the UC^{2}E, the *Simulation Data eXchange Protocol (SDXP)* has been specified. It enables the exchange of context information, control commands, and low-level hardware accesses required to perform power consumption estimations. Due to the defined protocols, the presented approach is independent of MDD tools and hardware platforms. Additionally, as an extension of software models,

a policy-oriented HAL has been introduced to abstract and encapsulate low-level hardware accesses during simulation. The HAL has been designed to be replaced by a platform-specific version in later phases of the MDD without affecting the software model.

The proposed power consumption estimation approach is evaluated based on a beehive microclimate IoT sensor node case study. As a proof-of-concept, the proposed developer workflow has been applied to the case study to demonstrate the potential of the overall approach. The evaluation includes the specification of software and hardware models, the elaboration of scenarios, the definition of power-related NFRs, an energy consumption estimation based on IPA and DPA, and the detection of software- and hardware-related energy bugs. For UML-based modeling and simulation, IBM Rhapsody has been selected as an MDD tool.

The results have shown that the concept of hardware component models and the PAP are suitable for modeling the energy-related behavior of hardware components. During simulation, the UC^{2}E tool was able to trace hardware accesses, create energy traces, and perform power consumption estimations. Along with scenario definitions, the energy traces have been successfully used to detect software and hardware energy bugs. In addition to the simulation-based estimation of IPA, DPA is capable of obtaining realistic measurements using a physical *Model-Testbed*. Due to the true behavior of peripheral devices, the measured values can be considered very accurate. By reproducing hardware-software interactions, the *edit-cross-compile-flash-debug* cycle can be avoided, allowing an estimation in early development phases. A detailed analysis of the DPA revealed a minimum delay of 30 ms as the lowest limit for the interaction between the UC^{2}E tool and the *Model-Testbed* to execute hardware-software interactions. While this delay is sufficient for the case study, it is large enough to become relevant if batches of messages need to be transmitted within a short time, e.g., when simultaneous hardware accesses are generated.

The presented contributions of this thesis may help developers and engineers to improve their understanding of power and energy consumption at the software level. Moreover, they may drive the development of energy-aware software applications by providing appropriate design patterns and methods for MDD to estimate the impact of software applications on embedded systems right at the beginning of the development process. The main idea to estimate the power consumption early during development may help to reduce the overall development costs and the time-to-market since additional optimization phases could be avoided. From a technical perspective, the novel concepts of energy-aware design patterns and the power consumption estimation approach offer sufficient potential and opportunities for further research.

8.2 Outlook

This thesis aims to specify energy-aware software applications and to estimate the power consumption of software applications for embedded systems. The presented concepts have proved to be promising while still offering the potential for future research, investigations, and improvements. Some suggestions for future research and open topics are:

Expanding the Design Pattern Catalog

Following the proposed design pattern identification process, future research will continue to identify and derive additional energy-aware software design patterns, e.g., for energy-aware user interfaces and energy harvesting. While this thesis focused on the energy-aware design

patterns that address the general power consumption, there may also be design patterns that aim to optimize the maximum electric current consumption of individual hardware components or the overall system. Currently, all energy-aware software design patterns of the catalog are considered as a stand-alone and isolated problem-solution pair. Thus, an additional aspect of future work is the elaboration of interrelations between energy-aware design patterns to build a pattern language.

Extension and Performance Optimization of the Power Estimation Approach

The proof-of-concept provided in this thesis shows the potential of the power estimation approach. Future work may focus on modeling energy sources (e.g., solar modules), energy harvesting, and battery models to refine the estimations and predict the operational lifetime of embedded systems equipped with specific energy sources. Additionally, the defined UML profile may be extended to consider aspects of energy bugs, enabling an automatic evaluation of NFRs in MDD. Future work may also focus on concepts to automate power consumption estimation and energy bug detection processes. Solutions such as appropriate energy-aware software design patterns should be proposed for affected parts of software applications that need to be revised or optimized as enhanced feedback for developers and engineers.

Independent energy models of communication interfaces may be developed to address power and timing overhead caused by the communication between hardware components. The evaluation revealed a delay of 30 ms for the communication between the UC^{2}E analysis tool and the *Model-Testbed*, which is considered sufficient for a proof-of-concept implementation. However, optimization potential exists on the software and hardware level to lower the latencies and increase the power consumption estimation performance, such as using binary formats for the data transfer or increasing transfer speeds. While considering the message structure defined by *Model-RPC*, first experiments using a binary message format yielded promising preliminary results with significant time delay reductions.

An important next step following the proof-of-concept might be the consideration of more extensive embedded systems. This also includes more complex communication interfaces like Wi-Fi and wired network connections with higher event-driven transmission activity. In both cases, it is expected that advanced environmental control will be required to predict the power consumption of peripheral devices more accurately.

It may be worth exploring other approaches, such as machine learning techniques, to derive energy models that describe the non-functional energy-related behavior of hardware components. Although approaches like [127] exist to generate energy models automatically, they are not able to capture hidden states and create relations between states. Furthermore, data obtained by the power consumption estimation approach may be used to train neural networks to predict the power and energy footprints of embedded systems.

Extend Support of MDD Tools and Functional Testing

Some of the findings in this thesis go beyond the intended scope of the proposed research questions and may be part of research in other areas. For instance, the DPA method may also be used for functional testing in early development phases or rapid prototyping approaches since the concept of *Model-Testbed* enables access to peripheral devices without executing the software application natively. Since the data exchange is based on actual data, e.g., obtained

by sensors, functional testing becomes possible. This also opens new research fields, e.g., the integration into MBT as a new approach for unit and system tests.

As part of the contributions, this thesis stated to provide a platform-independent approach. By defining communication protocols independent of specific MDD tools, it should be possible to use different MDD tools for the power consumption estimation approach. Ongoing work outside the scope of this thesis aims to adapt the proposed concept to include functional testing and advanced MDD language support. Preliminary results for MathWorks Matlab/Simulink using *Model-RPC* and *Model-Testbeds* to provide a rapid prototyping approach for functional testing based on realistic data have been submitted as new research and accepted as a scientific publication [393] at the *MODELS 2023* conference.

Bibliography

[1] 3rd Generation Partnership Project (3GPP). Cellular system support for ultra-low complexity and low throughput Internet of Things (CIoT). Technical report, 3GPP, 21 Dec. 2015. Document Number TR 45.820, V13.1.0.

[2] N. Abd El-Mawla, M. Badawy, and H. Arafat. IoT for the failure of climate-change mitigation and adaptation and IIoT as a future solution. *World Journal of Environmental Engineering*, 6(1):7–16, 2019. ISSN 2372-3076. doi: 10.12691/wjee-6-1-2.

[3] F. B. Abdallah and L. Apvrille. Fast evaluation of power consumption of embedded systems using diplodocus. In *39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 138–144, Santander, Spain, 4–6 Sept. 2013. doi: 10.1109/SEAA.2013.8.

[4] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong. Program energy efficiency: The impact of language, compiler and implementation choices. In *International Green Computing Conference*, IGCC '14, pages 1–6, Dallas, TX, USA, 3–5 Nov. 2014. ISBN 978-1-4799-6177-1. doi: 10.1109/IGCC.2014.7039169.

[5] Adafruit Industries. Adafruit BME280 Humidity + Barometric Pressure + Temperature Sensor Breakout, 2023. URL https://learn.adafruit.com/adafruit-bme280-humidity-barometric-pressure-temperature-sensor-breakout. Last Access: June 1st, 2023.

[6] W. Afzal, A. N. Ghazi, J. Itkonen, R. Torkar, A. Andrews, and K. Bhatti. An experiment on the effectiveness and efficiency of exploratory testing. *Empirical Software Engineering*, 20(3):844–878, June 2015. ISSN 1382-3256. doi: 10.1007/s10664-014-9301-4.

[7] D. Akdur, V. Garousi, and O. Demirörs. A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture*, 91:62–82, 2018. doi: 10.1016/j.sysarc.2018.09.007.

[8] F. Alam, P. R. Panda, N. Tripathi, N. Sharma, and S. Narayan. Energy optimization in android applications through wakelock placement. In *Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition*, DATE '14, pages 1–4, Dresden, Germany, 24–28 Mar. 2014. doi: 10.7873/DATE.2014.101.

[9] S. Albers and A. Antoniadis. Race to idle: New algorithms for speed scaling with a sleep state. *ACM Trans. Algorithms*, 10(2):1–31, Feb. 2014. ISSN 1549-6325. doi: 10.1145/2556953.

[10] C. Alexander, S. Ishikawa, and M. Silverstein. *A pattern language: Towns, buildings, construction*. Oxford Univ. Press, New York, NY, USA, 1977. ISBN 978-0-19-501919-3.

[11] A. Alexandrescu. *Modern C++ design: Generic programming and design patterns applied*. C++ in depth series. Addison-Wesley, Boston, MA, USA, 2nd edition, 2001. ISBN 978-0-201-70431-0.

[12] Altium Ltd. CircuitMaker, 2023. URL https://www.altium.com/circuitmaker. Last Access: February 1st, 2023.

[13] Amazon Web Services. FreeRTOS - market leading RTOS (real time operating system) for embedded systems with internet of things extensions, 2022. URL www.freertos.org/. Last Access: January 1st, 2023.

[14] D. Ameller. *Non-functional Requirements as Drivers of Software Architecture Design*. PhD thesis, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, Barcelona, Spain, 2014. URL http://hdl.handle.net/10803/144942.

[15] D. Ameller, X. Franch, C. Gómez, S. Martínez-Fernández, J. Araújo, S. Biffl, J. Cabot, V. Cortellessa, D. M. Fernández, A. Moreira, H. Muccini, A. Vallecillo, M. Wimmer, V. Amaral, W. Böhm, H. Bruneliere, L. Burgueño, M. Goulão, S. Teufl, and L. Berardinelli. Dealing with non-functional requirements in model-driven development: A survey. *IEEE Transactions on Software Engineering*, 47(4):818–835, 2021. doi: 10.1109/TSE.2019.2904476.

[16] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, 2nd edition, 2016. ISBN 978-1-107-17201-2. doi: 10.1017/9781316771273.

[17] ams OSRAM Group. *TSL2591 Light-to-Digital Converter*, 2023. URL https://ams.com/documents/20143/9331680/TSL2591_DS000338_7-00.pdf. Revision v3-00. Last Access: March 3rd, 2023.

[18] N. Amsel and B. Tomlinson. Green tracker: A tool for estimating the energy consumption of software. In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '10, pages 3337–3342, Atlanta, Georgia, USA, 2010. ISBN 978-1-60558-930-5. doi: 10.1145/1753846.1753981.

[19] E. Andrade, P. Maciel, T. Falcão, B. Nogueira, C. Araujo, and G. Callou. Performance and energy consumption estimation for commercial off-the-shelf component system design. *Innovations in Systems and Software Engineering*, 6(1-2):107–114, 2010. ISSN 1614-5046. doi: 10.1007/s11334-009-0110-7.

[20] Arm Limited. *CMSIS-Driver: Peripheral Interface for Middleware and Application Code - Version 2.8.0*, 2020. URL https://arm-software.github.io/CMSIS_5/latest/Driver/html/index.html. Last Access: June 1st, 2022.

[21] A. Armoush. *Design patterns for safety-critical embedded systems*. PhD thesis, RWTH Aachen University, Aachen, Germany, 2010. URL http://publications.rwth-aachen.de/record/51773.

[22] T. Arpinen, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen. Extension to MARTE profile for modeling dynamic power management of embedded systems. In *M-BED 1st Workshop on Model Based Engineering for Embedded Systems Design, Workhop co-located with DATE 2010*, pages 1–6, Dresden, Germany, 12 Mar. 2010.

[23] T. Arpinen, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen. MARTE profile extension for modeling dynamic power management of embedded systems. *Journal of Systems Architecture*, 58(5):209–219, Apr. 2012. ISSN 1383-7621. doi: 10.1016/j.sysarc.2011.01.003.

[24] A. Arrieta, I. Agirre, and A. Alberdi. Testing architecture with variability management in embedded distributed systems. In *Proceedings of the IV Jornadas de Computación Empotrada*, JCE '2013, pages 12–19, Madrid, Spain, 17–20 Sept. 2013.

[25] M. Asemani, F. Abdollahei, and F. Jabbari. Understanding IoT platforms : Towards a comprehensive definition and main characteristic description. In *Proceedings of the 5th International Conference on Web Research*, ICWR '19, pages 172–177, Tehran, Iran, 24–25 Apr. 2019. doi: 10.1109/ICWR.2019.8765259.

[26] AspenCore. 2019 embedded market study. Technical report, AspenCore, 2019. URL https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf. Last Access: September 9th, 2022.

[27] Y. B. Atitallah, J. Mottin, N. Hili, T. Ducroux, and G. Godet-Bar. A power consumption estimation approach for embedded software design using trace analysis. In *41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 61–68, Madeira, Portugal, 26–28 Aug. 2015. doi: 10.1109/SEAA.2015.34.

[28] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010. ISSN 1389-1286. doi: 10.1016/j.comnet.2010.05.010.

[29] J. Babić. *Model-Based Approach to Real-Time Embedded Control Systems Development with Legacy Components Integration.* PhD thesis, University of Zagreb - Faculty of Electrical Engineering and Computing, Zagreb, Croatia, 2014.

[30] M. Bagherzadeh, K. Jahed, B. Combemale, and J. Dingel. Live modeling in the context of state machine models and code generation. *Software & Systems Modeling*, 20(3): 795–819, 2021. ISSN 1619-1366. doi: 10.1007/s10270-020-00829-y.

[31] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile.* Springer Berlin, Heidelberg, 1st edition, 2008. ISBN 978-3-540-72562-6.

[32] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 280–293, Chicago, IL, USA, 4–6 Nov. 2009. ACM. ISBN 978-1-605-58771-4. doi: 10.1145/1644893.1644927.

[33] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 4 May 2004. doi: 10.1109/TSE.2004.9.

[34] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 588–598, 16–21 Nov. 2014. ISBN 978-1-450-33056-5. doi: 10.1145/2635868.2635871.

[35] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. On testing embedded software. volume 101 of *Advances in Computers*, chapter 3, pages 121–153. Elsevier, 2016. doi: 10.1016/bs.adcom.2015.11.005.

[36] A. Barkalov, L. Titarenko, and M. Mazurkiewicz. *Foundations of Embedded Systems*. Studies in Systems, Decision and Control. Springer International Publishing, Cham, Switzerland, 2019. ISBN 978-3-030-11960-7.

[37] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 1–33. Springer International Publishing, Cham, Switzerland, 2018. ISBN 978-3-319-75632-5. doi: 10.1007/978-3-319-75632-5_1.

[38] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley Professional, Boston, MA, USA, 4th edition, 2021. ISBN 978-0-13-688609-9.

[39] P. Baumann, M. Krammer, M. Driussi, L. Mikelsons, J. Zehetner, W. Mair, and D. Schramm. Using the distributed co-simulation protocol for a mixed real-virtual prototype. In *2019 IEEE International Conference on Mechatronics (ICM)*, volume 1, pages 440–445, 2019. doi: 10.1109/ICMECH.2019.8722844.

[40] E. Bayle, R. Bellamy, G. Casaday, T. Erickson, S. Fincher, B. Grinter, B. Gross, D. Lehder, H. Marmolin, B. Moore, C. Potts, G. Skousen, and J. Thomas. Putting it all together: Towards a pattern language for interaction design: A CHI 97 workshop. *ACM SIGCHI Bulletin*, 30(1):17–23, 1 Jan. 1998. ISSN 0736-6906. doi: 10.1145/280571.280580.

[41] K. Beck. *Smalltalk: Best Practice Patterns*. Prentice-Hall, Inc., Saddle River, NJ, USA, 1996. ISBN 978-0-13-476904-2.

[42] L. Benini, A. Bogliolo, and G. de Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, 2000. ISSN 1063-8210. doi: 10.1109/92.845896.

[43] H. Benninghoff, F. Rems, and T. Boge. Development and hardware-in-the-loop test of a guidance, navigation and control system for on-orbit servicing. *Acta Astronautica*, 102: 67–80, 2014. ISSN 0094-5765. doi: 10.1016/j.actaastro.2014.05.023.

[44] L. Berardinelli, P. Langer, and T. Mayerhofer. Combining fuml and profiles for non-functional analysis based on model execution traces. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13,

pages 79–88, Vancouver, BC, Canada, 17–21 June 2013. ISBN 978-1-4503-2126-6. doi: 10.1145/2465478.2465493.

[45] S. Berczuk. Finding solutions through pattern languages. *Computer*, 27(12):75–76, Dec. 1994. doi: 10.1109/2.335755.

[46] S. Bernardi and D. C. Petriu. Comparing two UML profiles for non-functional requirement annotations: the SPT and QoS profiles. In *International Workshop of Specification and Validation of UML models for Real Time and Embedded Systems in conjunction with the 7th International Conference on the Unified Modeling Language*, SVERTS '04, Lisbon, Portugal, 11 Oct. 2004.

[47] D.-R. Berte. Defining the IoT. *Proceedings of the International Conference on Business Excellence*, 12(1):118–128, May 2018. doi: 10.2478/picbe-2018-0013.

[48] J. Bézivin. In search of a basic principle for model-driven engineering. *Novatica Journal, Special Issue on UML*, 5(2):21–24, 2004.

[49] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston. *Object-oriented analysis and design with applications*. The Addison-Wesley object technology series. Addison-Wesley, Upper Saddle River, NJ, USA, 3rd edition, 2007. ISBN 978-0-201-89551-3.

[50] J. O. Borchers. A pattern approach to interaction design. *AI & Society*, 15(4):359–376, Dec. 2001. ISSN 0951-5666. doi: 10.1007/BF01206115.

[51] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228, Internet Engineering Task Force, May 2014. URL https://datatracker.ietf.org/doc/rfc7228/.

[52] Bosch Sensortec GmbH. *BMM160 – Datasheet, Version 1.4*, 2020. URL https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmm150-ds001.pdf. Last Access: August 3rd, 2022.

[53] Bosch Sensortec GmbH. *BME280 – Datasheet, Version 2.2*, 2021. URL https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf. Last Access: August 3rd, 2022.

[54] Bosch Sensortec GmbH. Github: BME280 sensor API. https://github.com/BoschSensortec/BME280_driver, 2022. URL https://github.com/BoschSensortec/BME280_driver. Last Access: August 3rd, 2022.

[55] T. Bouguera, J.-F. Diouris, J.-J. Chaillout, R. Jaouadi, and G. Andrieux. Energy consumption model for sensor nodes based on LoRa and LoRaWAN. *Sensors*, 18(7), 2018. ISSN 1424-8220. doi: 10.3390/s18072104.

[56] M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice*, volume 4 of *Synthesis lectures on software engineering*. Morgan & Claypool Publishers, San Rafael, CA, USA, 2nd edition, 2017. ISBN 978-1-62705-708-0. doi: 10.2200/S00751ED2V01Y201701SWE004.

[57] E. J. Braude and M. E. Bernstein. *Software Engineering: Modern Approaches*. Waveland Press, Long Grove, IL, USA, 2nd edition, 2016. ISBN 978-1-4786-3230-6.

[58] E. Bringmann and A. Krämer. Model-based testing of automotive systems. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, pages 485–493, Lillehammer, Norway, 9–11 Apr. 2008. doi: 10.1109/ICST.2008.45.

[59] B. Broekman and E. Notenboom. *Testing embedded software*. Pearson Education, 1st edition, 2003. ISBN 978-0-321-15986-1.

[60] D. Brown. A conversation with steve furber: The designer of the arm chip shares lessons on energy-efficient computing. *Queue*, 8(2):1–8, Feb. 2010. ISSN 1542-7730. doi: 10.1145/1716383.1716385.

[61] D. Bruggner, A. Hegde, F. S. Acerbo, D. Gulati, and T. D. Son. Model in the loop testing and validation of embedded autonomous driving algorithms. In *2021 IEEE Intelligent Vehicles Symposium (IV)*, pages 136–141, Nagoya, Japan, 11–17 July 2021. doi: 10.1109/IV48863.2021.9575530.

[62] J. Budde. Entwurf und Entwicklung einer Hardware-in-the-Loop Plattform zum Rapid-Prototyping modellbasierter Software. Master's thesis, Faculty of Engineering and Computer Science, Osnabrück University of Applied Sciences, Osnabrück, Germany, 2022. (in German).

[63] C. Bunse and H. Höpfner. Resource substitution with components - optimizing energy consumption. In *Proceedings of the 3rd International Conference on Software and Data Technologies*, volume 3 of *ICSOFT '08*, pages 28–35, Porto, Portugal, 5–8 July 2008. INSTICC. ISBN 978-989-8111-52-4. doi: 10.5220/0001879000280035.

[64] C. Bunse and S. Stiemer. On the energy consumption of design patterns. *Softwaretechnik-Trends*, 33(2):4–5, May 2013. doi: 10.1007/s40568-013-0020-6.

[65] T. D. Burd and R. W. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, ISLPED '00, pages 9–14, Rapallo, Italy, 26–27 July 2000. ACM. ISBN 978-1-58113-190-1. doi: 10.1145/344166.344181.

[66] M. Buschhoff, R. Falkenberg, and O. Spinczyk. Energy-aware device drivers for embedded operating systems. *ACM SIGBED Review*, 16(3):8–13, Nov. 2019. doi: 10.1145/3373400.3373401.

[67] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, Chichester, UK, 1996. ISBN 978-0-471-95869-7.

[68] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern Oriented Software Architecture: On Patterns and Pattern Languages (Wiley Software Patterns Series)*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2007. ISBN 978-0-471-48648-0.

[69] G. Caplat and J. L. Sourrouille. Model mapping in MDA. In *Workshop in Software Model Engineering at the 5th International Conference on the Unified Modeling Language*, volume 196 of *WiSME '02*, Dresden, Germany, 1 Oct. 2002.

[70] G. Caplat and J. L. Sourrouille. Considerations about model mapping. In J. Bezivin and M. Gogolla, editors, *Workshop in software model engineering at the 6th International Conference of The Unified Modeling Language*, WiSME '03. San Francisco, CA, USA, 21 Oct. 2003.

[71] I. Cassar and A. Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. In J. Cámara and J. Proença, editors, *Proceedings of the 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems*, volume 175 of *FOCLASA '14*, pages 54–68, Rome, Italy, 6 Sept. 2014. doi: 10.4204/EPTCS.175.4.

[72] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In A. Francalanza and G. J. Pace, editors, *Proceedings of the 2nd International Workshop on Pre- and Post-Deployment Verification Techniques*, volume 254 of *PrePost@iFM '17*, pages 15–28, Turin, Italy, 19 Sept. 2017. doi: 10.4204/EPTCS.254.2.

[73] M. Cattani, C. A. Boano, and K. Römer. An experimental evaluation of the reliability of lora long-range low-power wireless communication, 2017. ISSN 2224-2708. URL https://www.mdpi.com/2224-2708/6/2/7.

[74] M. R. V. Chaudron, W. Heijstek, and A. Nugroho. How effective is UML modeling? *Software & Systems Modeling*, 11(4):571–580, 26 Aug. 2012. ISSN 1619-1366. doi: 10.1007/s10270-012-0278-4.

[75] D. Cheij. A software architecture for building interchangeable test systems. In *2001 IEEE Autotestcon Proceedings. IEEE Systems Readiness Technology Conference*, pages 16–22, Valley Forge, PA, USA, 20–23 Aug. 2001. doi: 10.1109/AUTEST.2001.948916.

[76] M. Chen, J. Wan, and F. Li. Machine-to-machine communications: Architectures, standards and applications. *KSII Transactions on Internet and Information Systems*, 6 (2):480–497, 27 Feb. 2012. doi: 10.3837/tiis.2012.02.002.

[77] S. Chen, Y. Chen, S. Zhang, and N. Zheng. A novel integrated simulation and testing platform for self-driving cars with hardware in the loop. *IEEE Transactions on Intelligent Vehicles*, 4(3):425–436, 2019. doi: 10.1109/TIV.2019.2919470.

[78] P. S. Cheong, J. Bergs, C. Hawinkel, and J. Famaey. Comparison of lorawan classes and their power consumption. In *IEEE Symposium on Communications and Vehicular Technology*, SCVT '17, pages 1–6, 14 Nov. 2017. doi: 10.1109/SCVT.2017.8240313.

[79] K.-W. Choi and A. Chatterjee. Efficient instruction-level optimization methodology for low-power embedded systems. In *14th International Symposium on Systems Synthesis*, ISSS '01, pages 147–152, Montréal, P.Q., Canada, 30 Sept.–3 Oct. 2001. ISBN 978-1-58113-418-6. doi: 10.1145/500001.500035.

[80] Cisco Systems. Cisco annual internet report (2018–2023). Technical Report C11-741490-01, Cisco, 2020. URL https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Last Access: August 3rd, 2022.

[81] L. Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, Norwood, MA, USA, 2004. ISBN 978-1-58053-791-9.

[82] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, ISA, 1991. ISBN 978-0-201-54855-6.

[83] L. Corral, A. B. Georgiev, A. Sillitti, and G. Succi. A method for characterizing energy consumption in android smartphones. In *2nd International Workshop on Green and Sustainable Software (GREENS)*, pages 38–45, San Francisco, CA, USA, 20 May 2013. doi: 10.1109/GREENS.2013.6606420.

[84] V. Cortellessa, A. Di Marco, and P. Inverardi. Integrating performance and reliability analysis in a non-functional mda framework. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE '07, pages 57–71, Braga, Portugal, 24 Mar.–1 Apr. 2007.

[85] L. Cruz and R. Abreu. Performance-based guidelines for energy efficient mobile applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, pages 46–57, Buenos Aires, Argentina, 22–23 May 2017. doi: 10.1109/MOBILESoft.2017.19.

[86] L. Cruz and R. Abreu. Catalog of energy patterns for mobile applications. *Empirical Software Engineering*, 24(4):2209–2235, 2019. ISSN 1382-3256. doi: 10.1007/s10664-019-09682-0.

[87] A. Danese, G. Pravadelli, and I. Zandonà. Automatic generation of power state machines through dynamic mining of temporal assertions. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, DATE '16, pages 606–611, San Jose, CA, USA, 14–18 Mar. 2016. EDA Consortium. ISBN 978-3-98153706-2.

[88] A. Das, G. V. Merrett, and B. M. Al-Hashimi. The slowdown or race-to-idle question: Workload-aware energy optimization of smt multicore platforms under process variation. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, DATE '06, pages 535–538, 14–18 Mar. 2016.

[89] A. Dash, A. Kacker, C. Sutton, J. Zhou, and Y. Yoshida. Rational rhapsody JAVA API – code snippets & helper apps., 2022. URL https://www.ibm.com/support/pages/rational-rhapsody-java-api-\T1\textendash-code-snippets-helper-apps. Last Access: February 1st, 2023.

[90] Dassault Systèmes. Magicdraw, 2022. URL https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/. Last Access: August 3rd, 2022.

[91] M. de Miguel. General framework for the description of QoS in UML. In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 61–68, 16 May 2003. doi: 10.1109/ISORC.2003.1199237.

[92] J. Deichmann, D. Georg, B. Klein, B. Mühlreiter, and J. P. Stein. Cracking the complexity code in embedded systems development: How to manage - and eventually master - complexity in embedded systems development., 2022. URL https://www.mckinsey.com/industries/advanced-electronics/our-insights/

cracking-the-complexity-code-in-embedded-systems-development. Last Access: August 3rd, 2022.

[93] S. Dhouib, J.-P. Diguet, E. Senn, and J. Laurent. Energy models of real time operating systems on FPGA. In *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, Prague, Czech Republic, 1 July 2008.

[94] S. Dhouib, E. Senn, J.-P. Diguet, J. Laurent, and D. Blouin. Model driven high-level power estimation of embedded operating systems communication services. In *Proceedings of the 6th International Conference on Embedded Software and Systems*, pages 475–481, Hangzhou, ZJ, China, 25–27 May 2009. doi: 10.1109/ICESS.2009.94.

[95] A. C. Dias-Neto and G. H. Travassos. Model-based testing approaches selection for software projects. *Information and Software Technology*, 51(11):1487–1504, 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2009.06.010.

[96] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: A systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, WEASELTech '07, pages 31–36, Atlanta, Georgia, USA, 5 Nov. 2007. ACM. doi: 10.1145/1353673.1353681.

[97] B. Dobing and J. Parsons. Dimensions of UML diagram use. *Journal of Database Management*, 19(1):1–18, 2008. ISSN 1063-8016. doi: 10.4018/jdm.2008010101.

[98] B. P. Douglass. *Real-time design patterns: Robust scalable architecture for Real-time systems*. The Addison-Wesley object technology series. Addison-Wesley, Boston, MA, USA, 2003. ISBN 978-0-201-69956-2.

[99] B. P. Douglass. *Design patterns for embedded systems in C: An embedded software engineering toolkit*. Newnes/Elsevier, Oxford and Burlington, MA, 2011. ISBN 978-1-85617-707-8.

[100] B. P. Douglass. *Real-time UML workshop for embedded systems*. Embedded technology series. Newnes an imprint of Elsevier, Amsterdam, second edition edition, 2014. ISBN 978-0-12-407781-2.

[101] L.-T. Duan, B. Guo, Y. Shen, Y. Wang, and W.-L. Zhang. Energy analysis and prediction for applications on smartphones. *Journal of Systems Architecture*, 59(10, Part D):1375–1382, 2013. ISSN 1383-7621. doi: 10.1016/j.sysarc.2013.08.011.

[102] U. Durak, D. Müller, F. Möcke, and C. B. Koch. Modeling and simulation based development of an enhanced ground proximity warning system for multicore targets. In *Proceedings of the Model-Driven Approaches for Simulation Engineering Symposium*, Mod4Sim '18, Baltimore, Maryland, 15–18 Apr. 2018. Society for Computer Simulation International.

[103] Y. Durrani, T. Riesgo, and F. Machado. Statistical power estimation for register transfer level. In *International Conference Mixed Design of Integrated Circuits and System*, pages 522–527, Gdynia, Poland, 22–24 June 2006. doi: 10.1109/MIXDES.2006.1706635.

[104] Eclipse Foundation. Papyrus, 2022. URL https://www.eclipse.org/papyrus. Last Access: August 3rd, 2022.

[105] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow, and M. N. Hindia. An overview of internet of things (IoT) and data analytics in agriculture: Benefits and challenges. *IEEE Internet of Things Journal*, 5(5):3758–3773, 2018. doi: 10.1109/JIOT.2018.2844296.

[106] I. Eouzan, L. Garnery, M. A. Pinto, D. Delalande, C. J. Neves, F. Fabre, J. Lesobre, S. Houte, A. Estonba, I. Montes, T. Sime-Ngando, and D. G. Biron. Hygroregulation, a key ability for eusocial insects: Native western european honeybees as a case study. *PLOS ONE*, 14(2):1–15, Feb. 2019. doi: 10.1371/journal.pone.0200048.

[107] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *38th Annual International Symposium on Computer Architecture*, volume 39 of *ISCA '11*, pages 365–376, San Jose, CA, USA, 4–8 June 2011. doi: 10.1145/2024723.2000108.

[108] H. Espinoza, H. Dubois, S. Gérard, J. Medina, D. C. Petriu, and M. Woodside. Annotating uml models with non-functional properties for quantitative analysis. In *Proceedings of the 2005 International Conference on Satellite Events at the 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '05, pages 79–90, Montego Bay, Jamaica, 2–7 Oct. 2005. Springer-Verlag. ISBN 978-3-54031780-7. doi: 10.1007/11663430_9.

[109] H. Espinoza, H. Dubois, J. Medina, and S. Gérard. A general structure for the analysis framework of the UML MARTE profile. In *MARTES Workshop at 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '05, Montego Bay, Jamaica, 4 Oct. 2005.

[110] H. Espinoza, J. Medina, H. Dubois, S. Gérard, and F. Terrier. Towards a UML-based modelling standard for schedulability analysis of real-time systems. In *MARTES Workshop at 9th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '06, pages 79–90, Genoa, Italy, 2 Oct. 2006.

[111] Espressif Systems. *ESP32 Series*, 2022. URL https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. Last Access: August 3rd, 2022.

[112] Espressif Systems. Esp-idf - iot development framework, 2023. URL https://www.espressif.com/en/products/sdks/esp-idf. Last Access: February 3rd, 2023.

[113] ETSI Technical Committee Methods for Testing and Specification. Methods for testing and specification (MTS); model-based testing (MBT); requirements for modelling notations. Standard ES 202 951 V1.1.1, European Telecommunications Standards Institute, Sophia Antipolis, France, 2011. URL https://www.etsi.org/deliver/etsi_es/202900_202999/202951/01.01.01_60/es_202951v010101p.pdf.

[114] E. Evans and E. J. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. ISBN 978-0-321-12521-7.

[115] EventHelix.com Inc. High speed serial port design pattern, 2019. URL https://www.eventhelix.com/design-patterns/high-speed-serial-port/. Last Access: August 3rd, 2020.

[116] Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy, D. A. Peled, and G. Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013. doi: 10.3233/978-1-61499-207-3-141.

[117] Y. Falcone, S. Krstić, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. *International Journal on Software Tools for Technology Transfer*, 23 (2):255–284, 2021. ISSN 1433-2787. doi: 10.1007/s10009-021-00609-z.

[118] Z. S. Fathy, Hosam K.and Filipi, J. Hagena, and J. L. Stein. Review of hardware-in-the-loop simulation and its prospects in the automotive area. In K. Schum and A. F. Sisti, editors, *Modeling and Simulation for Military Applications*, volume 6228, pages 117–136, Orlando, FL, USA, 22 May 2006. International Society for Optics and Photonics, SPIE. doi: 10.1117/12.667794.

[119] M. Faugere, T. Bourbeau, R. d. Simone, and S. Gerard. MARTE: Also an UML profile for modeling AADL applications. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, ICECCS '07, pages 359–364, Auckland, New Zealand, 11–14 July 2007. doi: 10.1109/ICECCS.2007.29.

[120] C. Fehling, J. Barzen, U. Breitenbücher, and F. Leymann. A process for pattern identification, authoring, and application. In *Proceedings of the 19th European Conference on Pattern Languages of Programs*, EuroPLoP '14, 9–13 July 2014. doi: 10.1145/2721956.2721976.

[121] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud Computing Patterns*. Springer Vienna, Vienna, Austria, 2014. ISBN 978-3-7091-1567-1. doi: 10.1007/978-3-7091-1568-8.

[122] D. Feitosa, R. Alders, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa. Investigating the effect of design patterns on energy consumption. *Journal of Software: Evolution and Process*, 29(2):e1851, 2017. doi: 10.1002/smr.1851.

[123] A. Fonseca, R. Kazman, and P. Lago. A manifesto for energy-aware software. *IEEE Software*, 36(6):79–82, 2019. doi: 10.1109/MS.2019.2924498.

[124] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, Boston, MA, USA, 1st edition, 2003. ISBN 978-0-321-12742-6.

[125] M. Fowler. *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley, Boston, MA, USA, 3rd edition, 2004. ISBN 978-0-321-19368-1.

[126] M. Friedli, L. Kaufmann, F. Paganini, and R. Kyburz. Energy efficiency of the internet of things. *Technology and Energy Assessment Report prepared for IEA 4E EDNA. Lucerne University of Applied Sciences, Switzerland*, 2016.

[127] D. Friesel, M. Buschhoff, and O. Spinczyk. Parameter-aware energy models for embedded-system peripherals. In *Proceedings of the 13th International Symposium on Industrial Embedded Systems*, SIES '18, pages 1–4, Graz, Austria, 6–8 June 2018. doi: 10.1109/SIES.2018.8442096.

[128] T. Funk and B. Wicht. *Integrated Wide-Bandwidth Current Sensing*. Springer International Publishing, Cham, Switzerland, 2020. ISBN 978-3-030-53249-9. doi: 10.1007/978-3-030-53250-5.

[129] J. Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16 (3):225–233, 1986. doi: 10.1002/spe.4380160304.

[130] D. D. Gajski and R. H. Kuhn. Guest editors' introduction: New VLSI tools. *Computer*, 16(12):11–14, 1983. doi: 10.1109/MC.1983.1654264.

[131] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, MA, USA, 1994. ISBN 978-0-201-63361-0.

[132] J. Ganssle and M. Barr. *Embedded Systems Dictionary*. CMP Books, San Francisco, CA, USA, 2003. ISBN 978-1-57820-120-4.

[133] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz. Testing embedded software: A survey of the literature. *Information and Software Technology*, 104:14–45, 2018. ISSN 0950-5849. doi: 10.1016/j.infsof.2018.06.016.

[134] Ş. Y. Gelbal, S. Tamilarasan, M. R. Cantaş, L. Güvenç, and B. Aksun-Güvenç. A connected and autonomous vehicle hardware-in-the-loop simulator for developing automated driving algorithms. In *Proceedings of the 2017 IEEE International Conference on Systems, Man, and Cybernetics*, SMC '17, pages 3397–3402, Banff, AB, Canada, 5–8 Oct. 2017. doi: 10.1109/SMC.2017.8123155.

[135] K. Georgiou, S. Xavier-de Souza, and K. Eder. The IoT energy challenge: A software perspective. *IEEE Embedded Systems Letters*, 10(3):53–56, 2018. doi: 10.1109/LES.2017.2741419.

[136] S. Georgiou, M. Kechagia, P. Louridas, and D. Spinellis. What are your programming language's energy-delay implications? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 303–313, 27 May–3 June 2018. doi: 10.1145/3196398.3196414.

[137] M. Glinz. On non-functional requirements. In *Proceedings of the 15th IEEE International Requirements Engineering Conference*, RE '07, pages 21–26, Delhi, India, 15–19 Oct. 2007. doi: 10.1109/RE.2007.45.

[138] H. Gomaa. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, Cambridge, UK, 2011. ISBN 978-0-521-76414-8.

[139] Google LLC. Protocol buffers. Technical report, Google LLC, 2021. URL https://developers.google.com/protocol-buffers. Last Access: August 3rd, 2022.

[140] Google LLC. gRPC: A high-performance, open source universal RPC framework, 2022. URL https://grpc.io/. Last Access: August 3rd, 2022.

[141] Google LLC. Flatbuffers, 2022. URL https://google.github.io/flatbuffers/. Last Access: August 3rd, 2022.

[142] N. Goumagias, J. Whalley, O. Dilaver, and J. Cunningham. Making sense of the internet of things: a critical review of internet of things definitions between 2005 and 2019. *Internet Research*, 31(5):1583–1610, 2021. ISSN 1066-2243. doi: 10.1108/INTR-01-2020-0013.

[143] M. D. Grammatikakis, G. Kornaros, and M. Coppola. Power–aware multicore soc and noc design. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, pages 167–193. Springer New York, New York, NY, USA, 2011. ISBN 978-1-4419-6460-1. doi: 10.1007/978-1-4419-6460-1_8.

[144] R. Grønmo and B. Møller-Pedersen. From UML 2 sequence diagrams to state machines by graph transformation. *Journal of Object Technology*, 10:8: 1–22, 2011. doi: 10.5381/jot.2011.10.1.a8.

[145] D. Gross and E. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001. ISSN 0947-3602. doi: 10.1007/s007660170013.

[146] A. Grunwald, M. Schaarschmidt, and C. Westerkamp. LoRaWAN in a rural context: Use cases and opportunities for agricultural businesses. In P. Roer, editor, *Proceedings of the Mobile Communication-Technologies and Applications; 24. ITG-Symposium*, ITG-Fachbericht, pages 134–139. VDE-Verlag GmbH, Osnabrück, Germany, 15–16 May 2019.

[147] I. Gräßler, J. Hentze, and T. Bruckmann. V-models for interdisciplinary systems engineering. In D. Marjanović, M. Štorga, S. Škec, N. Bojčetić, and N. Pavković, editors, *Proceedings of the 15th International Design Conference*, pages 747–756, Dubrovnik, Croatia, 21–24 May 2018. doi: 10.21278/idc.2018.0333.

[148] L. Gui, J. Sun, Y. Liu, Y. J. Si, J. S. Dong, and X. Y. Wang. Combining model checking and testing with an application to reliability prediction and distribution. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, pages 101–111, Lugano, Switzerland, 15–20 July 2013. ACM. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483779.

[149] P. Guo, Y. Li, P. Li, S. Liu, and D. Sun. A uml model to simulink model transformation method in the design of embedded software. In *Proceedings of the 2014 Tenth International Conference on Computational Intelligence and Security*, pages 583–587, Kunming, China, 15–16 Nov. 2014. doi: 10.1109/CIS.2014.162.

[150] J. D. Hagar. *Software Test Attacks to Break Mobile and Embedded Devices*. Chapman & Hall/CRC, Boca Raton, FL, USA, 1st edition, 2017. ISBN 978-1-138-46844-3.

[151] M. Hagner, A. Aniculaesei, and U. Goltz. UML-based analysis of power consumption for real-time embedded systems. In *Proceedings of the 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1196–1201, Changsha, HN, China, 16–18 Nov. 2011. IEEE. ISBN 978-1-4577-2135-9. doi: 10.1109/TrustCom.2011.161.

[152] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006. doi: 10.1147/sj.453.0451.

[153] J. Hansson, S. Helton, and P. Feiler. ROI analysis of the system architecture virtual integration initiative. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 2018.

[154] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE' 13, pages 92–101, San Francisco, CA, USA, 18–26 May 2013. doi: 10.1109/ICSE.2013.6606555.

[155] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. ISSN 0167-6423. doi: 10.1016/0167-6423(87)90035-9.

[156] K. Havelund and A. Goldberg. Verify your runs. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, Revised Selected Papers and Discussions*, VSTTE '05, pages 374–383. Springer Berlin Heidelberg, Berlin, Germany, 10–13 Oct. 2008. ISBN 978-3-540-69149-5. doi: 10.1007/978-3-540-69149-5_40.

[157] D. Helms, E. Schmidt, and W. Nebel. Leakage in CMOS circuits – an introduction. In E. Macii, V. Paliouras, and O. Koufopavlou, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 17–35, Berlin, Heidelberg, 15–17 Sept. 2004. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-30205-6_5.

[158] T. Hermans, P. Ramaekers, J. Denil, P. D. Meulenaere, and J. Anthonis. Incorporation of AUTOSAR in an embedded systems development process: A case study. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 247–250, Oulu, Finland, 30 Aug.–2 Sept. 2011. doi: 10.1109/SEAA.2011.45.

[159] G. Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, Boston, MA, USA, 1st edition edition, 2003. ISBN 978-0-321-20068-6.

[160] T. Hönig. *Proactive Energy-Aware Computing*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany, 2017. URL https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/8992.

[161] T. Hönig, H. Janker, C. Eibel, W. Schröder-Preikschat, O. Mihelic, and R. Kapitza. Proactive energy-aware programming with peek. In *Proceedings of the 2014 International Conference on Timely Results in Operating Systems*, TRIOS '14, pages 1–14, Broomfield, CO, USA, 5 Oct. 2014. USENIX Association.

[162] L. Huning and E. Pulvermüller. Automatic code generation of safety mechanisms in model-driven development. *Electronics*, 10(24), 2021. ISSN 2079-9292. doi: 10.3390/electronics10243150. URL https://www.mdpi.com/2079-9292/10/24/3150.

[163] L. Huning, T. Osterkamp, M. Schaarschmidt, and E. Pulvermüller. Seamless integration of hardware interfaces in UML-based MDSE tools. In *Proceedings of the 16th International Conference on Software Technologies*, volume 1 of *ICSOFT '21*, pages 233–244,

Online Streaming, 6–8 July 2021. INSTICC, SciTePress. ISBN 978-989-758-523-4. doi: 10.5220/0010575802330244.

[164] IBM. IBM Engineering Systems Design Rhapsody - Developer, 2022. URL https://www.ibm.com/products/uml-tools. Last Access: August 3rd, 2022.

[165] IBM. IBM Docs: Model Based Testing with TestConductor and Automatic Test Generation (ATG), 2022. URL https://www.ibm.com/docs/en/rhapsody/9.0.1?topic=dm-model-based-testing-testconductor-automatic-test-generation-atg. Last Access: August 3rd, 2022.

[166] IBM. IBM Documentation - Helpers, 2023. URL https://www.ibm.com/docs/en/rhapsody/9.0.1?topic=rhapsody-helpers. Last Access: February 1st, 2023.

[167] IBM. IBM Documentation - Rhapsody API, 2023. URL https://www.ibm.com/docs/en/rhapsody/9.0.1?topic=function-rhapsody-api. Last Access: February 1st, 2023.

[168] IBM. IBM Documentation - Generating code for component diagrams, 2023. URL https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/design-rhapsody/9.0.2?topic=code-generating-component-diagrams. Last Access: February 1st, 2023.

[169] Institute of Electrical and Electronics Engineers, Inc. IEEE standard american national standard canadian standard graphic symbols for electrical and electronics diagrams (including reference designation letters). Technical report, 1993.

[170] Institute of Electrical and Electronics Engineers, Inc. IEEE standard for standard systemc language reference manual. Technical report, Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, 2012.

[171] Institute of Electrical and Electronics Engineers, Inc. IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. Technical report, Institute of Electrical and Electronics Engineers, 2014. URL https://standards.ieee.org/standard/1685-2014.html. Last Access: August 3rd, 2022.

[172] International Software Testing Qualifications Board. ISTQB Glossary, 2022. URL https://glossary.istqb.org/. Last Access: August 3rd, 2022.

[173] R. Isermann, J. Schaffnit, and S. Sinsel. Hardware-in-the-loop simulation for the design and testing of engine-control systems. *IFAC Proceedings Volumes*, 31(4):1–10, 15–17 Apr. 1998. ISSN 1474-6670. doi: 10.1016/S1474-6670(17)42125-2. 5th IFAC Workshop on Algorithms & Architecture for Real Time Control.

[174] ISO, IEC, and IEEE. ISO/IEC/IEEE international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, 2017. doi: 10.1109/IEEESTD.2017.8016712.

[175] J. Itkonen and K. Rautiainen. Exploratory testing: a multiple case study. In *2005 International Symposium on Empirical Software Engineering*, ISESE '05, Noosa Heads, Queensland, Australia, 17–18 Nov. 2005. doi: 10.1109/ISESE.2005.1541817.

[176] IVI Foundation. Standard Commands for Programmable Instruments (SCPI). Technical report, European SCPI Consortium, 1999. URL https://www.ivifoundation.org/docs/scpi-99.pdf. Document Number SCPI-99. Last Access: August 3rd, 2022.

[177] IVI Foundation. *VISA Specifications*, 2022. URL https://www.ivifoundation.org/downloads/ArchitectureSpecifications/vpp43_2022-05-19.pdf. Document Number VPP-4.3, Revision 7.2. Last Access: August 3rd, 2022.

[178] P. Iyenghar and E. Pulvermüller. A model-driven workflow for energy-aware scheduling analysis of IoT-enabled use cases. *IEEE Internet of Things Journal*, 5(6):4914–4925, 2018. doi: 10.1109/JIOT.2018.2879746.

[179] P. Iyenghar, A. Noyer, J. Engelhardt, and E. Pulvermueller. Translating timing requirements of embedded software systems modeled in Simulink to a timing analysis model. In *Proceedings of the 21st International Conference on Emerging Technologies and Factory Automation*, ETFA '16, pages 1–4, Berlin, Germany, 6–9 Sept. 2016. doi: 10.1109/ETFA.2016.7733662.

[180] P. Iyenghar, A. Noyer, J. Engelhardt, and E. Pulvermueller. Model-based co-engineering and NFP analysis in embedded software sub-systems developed using heterogeneous modeling domains. In *Proceedings of the 14th International Conference on Industrial Informatics*, INDIN '2016, pages 1154–1161, Poitiers, France, 19–21 July 2016. doi: 10.1109/INDIN.2016.7819340.

[181] P. Iyenghar, A. Noyer, J. Engelhardt, E. Pulvermueller, and C. Westerkamp. End-to-end path delay estimation in embedded software involving heterogeneous models. In *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems*, SIES '16, pages 1–6, Krakow, Poland, 23–25 May 2016. doi: 10.1109/SIES.2016.7509427.

[182] P. Iyenghar, S. Wessels, A. Noyer, E. Pulvermueller, and C. Westerkamp. A novel approach towards model-driven reliability analysis of Simulink models. In *Proceedings of the 21st International Conference on Emerging Technologies and Factory Automation*, ETFA '16, pages 1–6, Berlin, Germany, 6–9 Sept. 2016. doi: 10.1109/ETFA.2016.7733505.

[183] P. Iyenghar, A. Noyer, and E. Pulvermüller. Early model-driven timing validation of IoT-compliant use cases. In *Proceedings of the 15th International Conference on Industrial Informatics*, INDIN '17, pages 19–25, Emden, Germany, 24–26 July 2017. doi: 10.1109/INDIN.2017.8104740.

[184] P. M. Jacob and M. Prasanna. A comparative analysis on black box testing strategies. In *Proceedings of the 2016 International Conference on Information Science*, ICIS '16, pages 1–6, Kochi, India, 12–13 Aug. 2016. doi: 10.1109/INFOSCI.2016.7845290.

[185] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley, Reading, Mass. and Harlow, 1999. ISBN 978-0-201-57169-1.

[186] V. Jaikamal. Model-based ECU development – an integrated MiL-SiL-HiL approach. In *SAE World Congress & Exhibition*, Detroit, MI, USA, 20 Apr. 2009. SAE International. doi: 10.4271/2009-01-0153.

[187] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan. Powering the internet of things. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pages 375–380, La Jolla, California, USA, 11–13 Aug. 2014. doi: 10.1145/2627369.2631644.

[188] H. Jiang, M. Marek-Sadowska, and S. R. Nassif. Benefits and costs of power-gating technique. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pages 559–566, San Jose, CA, USA, 2–5 Oct. 2005. IEEE Computer Society. ISBN 978-0-7695-2451-1. doi: 10.1109/ICCD.2005.34.

[189] P. C. Jorgensen. *Software Testing: a Craftsman's Approach*. CRC Press, Boca Raton, FL, USA, 4th edition, 2014. ISBN 978-1-4665-6069-7.

[190] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*, 2013. URL https://www.jsonrpc.org/specification. Last Access: August 3rd, 2022.

[191] N. Julien, J. Laurent, E. Senn, and E. Martin. Power estimation of a C algorithm based on the functional-level power analysis of a digital signal processor. In *Proceedings of the 4th International Symposium on High Performance Computing*, ISHPC '02, pages 354–360, Kansai Science City, Japan, 15–17 May 2002. Springer-Verlag. ISBN 978-3-54043674-4. doi: 10.1007/3-540-47847-7_32.

[192] M. Kallmann and D. Thalmann. Modeling objects for interaction tasks. In B. Arnaldi and G. Hégron, editors, *Computer Animation and Simulation '98*, pages 73–86. Springer Vienna, Vienna, Austria, 1999. ISBN 978-3-7091-6375-7.

[193] D. Kamma and G. S. Kumar. Effect of model based software development on productivity of enhancement tasks – an industrial study. In *21st Asia-Pacific Software Engineering Conference*, volume 1, pages 71–77, 1–4 Dec. 2014. doi: 10.1109/APSEC.2014.20.

[194] A. Kanduri, A. M. Rahmani, P. Liljeberg, A. Hemani, A. Jantsch, and H. Tenhunen. A perspective on dark silicon. In A. M. Rahmani, P. Liljeberg, A. Hemani, A. Jantsch, and H. Tenhunen, editors, *The Dark Side of Silicon: Energy Efficient Computing in the Dark Silicon Era*, pages 3–20. Springer International Publishing, Cham, Switzerland, 2017. ISBN 978-3-319-31596-6. doi: 10.1007/978-3-319-31596-6_1.

[195] S. Kaxiras and M. Martonosi. *Computer architecture techniques for power-efficiency*, volume 4 of *Synthesis lectures on computer architecture*. Morgan & Claypool, San Rafael, CA, USA, 2008. ISBN 978-1-59829-208-4.

[196] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi. *Low Power Methodology Manual - For System-on-Chip Design*. Springer Publishing, New York, NY, USA, 2007. ISBN 978-0-387-71818-7. doi: 10.1007/978-0-387-71819-4.

[197] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 338–349, 14–16 July 1997. doi: 10.1109/ASAP.1997.606839.

[198] B. Kienhuis, E. F. Deprettere, P. v. d. Wolf, and K. A. Vissers. A methodology to design programmable embedded systems - the Y-chart approach. In *Embedded Processor*

Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS, pages 18–37, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 978-3-54043322-4.

[199] D.-H. Kim, J.-P. Kim, and J.-E. Hong. A power consumption analysis technique using UML-based design models in embedded software development. In I. Černá, T. Gyimóthy, J. Hromkovič, K. Jefferey, R. Královič, M. Vukolić, and S. Wolf, editors, *SOFSEM 2011: Theory and Practice of Computer Science*, volume 6543 of *Lecture Notes in Computer Science*, pages 320–331. Springer, Berlin, Germany, 2011. ISBN 978-3-642-18380-5. doi: 10.1007/978-3-642-18381-2_27.

[200] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 19 Dec. 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1250885.

[201] Kingbright. *AP2012EC – Datasheet, Version 11A*, 2021. URL https://www.kingbrightusa.com/images/catalog/SPEC/AP2012EC.pdf. Last Access: August 3rd, 2022.

[202] J. C. Knight. Dependability of embedded systems. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 685–686, Orlando, FL, USA, 19–25 May 2002. ISBN 978-1-58113-472-8. doi: 10.1145/581339.581445.

[203] C. B. Koch, U. Durak, and D. Müller. Simulation-based verification for parallelization of model-based applications. In *Proceedings of the 50th Computer Simulation Conference*, SummerSim '18, Bordeaux, France, 9–12 July 2018. Society for Computer Simulation International.

[204] S. Konrad and B. H. C. Cheng. Requirements patterns for embedded systems. In *Proceedings IEEE Joint International Conference on Requirements Engineering*, pages 127–136, 9–13 Sept. 2002. doi: 10.1109/icre.2002.1211541.

[205] S. Konrad, B. H. C. Cheng, and L. A. Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970–992, 2004. doi: 10.1109/TSE.2004.102.

[206] V. Konstantakos, A. Chatzigeorgiou, S. Nikolaidis, and T. Laopoulos. Energy consumption estimation in embedded systems. *IEEE Transactions on Instrumentation and Measurement*, 57(4):797–804, 5 Mar. 2008. doi: 10.1109/TIM.2007.913724.

[207] P. Koopman. Embedded system security. *Computer*, 37(7):95–97, 12 July 2004. doi: 10.1109/MC.2004.52.

[208] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton. Smart objects as building blocks for the internet of things. *IEEE Internet Computing*, 14(1):44–51, 1 Dec. 2010. doi: 10.1109/MIC.2009.143.

[209] M. Krammer, C. Schiffer, and M. Benedikt. ProMECoS: A process model for efficient standard-driven distributed co-simulation. *Electronics*, 10(5), 2021. ISSN 2079-9292. doi: 10.3390/electronics10050633.

[210] Y. Kuroki, M. Yoo, and T. Yokoyama. A Simulink to UML model transformation tool for embedded control software development. In *Proceedings of the 2016 IEEE*

International Conference on Industrial Technology, ICIT '26, pages 700–706, Taipei, Taiwan, 14–17 Mar. 2016. doi: 10.1109/ICIT.2016.7474835.

[211] H. J. Landau. Sampling, data transmission, and the Nyquist rate. *Proceedings of the IEEE*, 55(10):1701–1706, 1967. ISSN 0018-9219. doi: 10.1109/PROC.1967.5962.

[212] O. Landsiedel, K. Wehrle, and S. Gotz. Accurate prediction of power consumption in sensor networks. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, EmNetS-II '05, pages 37–44, 31 May 2005. doi: 10.1109/EMNETS.2005.1469097.

[213] C. Lange, M. Chaudron, and J. Muskens. In practice: UML software architecture and design description. *IEEE Software*, 23(2):40–46, 13 Mar. 2006. doi: 10.1109/MS.2006.50.

[214] P. A. Laplante. *Requirements Engineering for Software and Systems*. CRC Press, Boca Raton, FL, USA, 3rd edition, 2017. ISBN 978-1-138-19611-7.

[215] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 19 June 2014. doi: 10.1007/s12599-014-0334-4.

[216] K.-K. Lau, F. M. Taweel, and C. M. Tran. The W model for component-based software development. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '11, pages 47–50, Oulu, Finland, 30 Aug.–2 Sept. 2011. doi: 10.1109/SEAA.2011.17.

[217] J. Laurent, E. Senn, N. Julien, and E. Martin. High level energy estimation for DSP systems. In *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation*, PATMOS '01, pages 311–316, Yverdon-les-Bains, Switzerland, 26–28 Sept. 2001.

[218] P. Lea. *Internet of Things for Architects: Architecting IoT solutions by implementing sensors, communication infrastructure, edge computing, analytics, and security*. Packt Publishing, Birmingham, UK, 1st edition, 2018. ISBN 978-1-78847-059-9.

[219] G. M. Lee, N. Crespi, J. K. Choi, and M. Boussard. Internet of things. In E. Bertin, N. Crespi, and T. Magedanz, editors, *Evolution of Telecommunication Services: The Convergence of Telecom and Internet: Technologies and Ecosystems*, pages 257–282, Berlin, Germany, 2013. Springer Verlag. ISBN 978-3-642-41569-2. doi: 10.1007/978-3-642-41569-2_13.

[220] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. ISSN 1567-8326. doi: 10.1016/j.jlap.2008.08.004.

[221] D. Li, S. Hao, J. Gui, and W. G. Halfond. An empirical study of the energy consumption of android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 121–130, Victoria, BC, Canada, 29 Sept.–3 Oct. 2014. doi: 10.1109/ICSME.2014.34.

[222] P. Liggesmeyer and M. Trapp. Trends in embedded software engineering. *IEEE Software*, 26(3):19–25, 2009. doi: 10.1109/MS.2009.80.

[223] C. Lim, H. T. Ahn, and J. T. Kim. Predictive DVS scheduling for low-power real-time operating system. In *International Conference on Convergence Information Technology (ICCIT)*, pages 1918–1921, Gwangju, Korea, 21–23 Nov. 2007. IEEE Computer Society. ISBN 978-0-7695-3038-3. doi: 10.1109/ICCIT.2007.316.

[224] A. Litke, K. Zotos, A. Chatzigeorgiou, and G. Stephanides. Energy consumption analysis of design patterns. *International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering*, 1(11):1663–1667, 2007.

[225] Y. Liu, L. Gui, and Y. Liu. MDP-based reliability analysis of an ambient assisted living system. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 688–702, Cham, Switzerland, 12–16 May 2014. Springer International Publishing. doi: 10.1007/978-3-319-06410-9_46.

[226] Y. Liu, C. Xu, and S.-C. Cheung. Diagnosing energy efficiency and performance for mobile internetware applications. *IEEE Software*, 32(1):67–75, 2015. doi: 10.1109/MS.2015.4.

[227] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 396–409, Seattle, WA, USA, 13–18 Nov. 2016. doi: 10.1145/2950290.2950297.

[228] Y. D. Liu. Energy-efficient synchronization through program patterns. In *Proceedings of the 1st International Workshop on Green and Sustainable Software in conjunction with the 34th International Conference on Software Engineering*, GREENS '12, pages 35–40, Zurich, Switzerland, 3 June 2012. doi: 10.1109/GREENS.2012.6224253.

[229] D. Lopes, S. Hammoudi, J. Bézivin, and F. Jouault. Mapping specification in MDA: From theory to practice. In D. Konstantas, J.-P. Bourrières, M. Léonard, and N. Boudjlida, editors, *Interoperability of Enterprise Software and Applications*, pages 253–264. Springer London, London, UK, 2006. ISBN 978-1-84628-151-8. doi: 10.1007/1-84628-152-0_23.

[230] Lora Alliance. *LoRaWAN™ 1.1 Specification*, 2017. URL https://lora-alliance.org/wp-content/uploads/2020/11/lorawantm_specification_-v1.1.pdf. Last Access: August 3rd, 2022.

[231] G. Luo, B. Guo, Y. Shen, H. Liao, and L. Ren. Analysis and optimization of embedded software energy consumption on the source code and algorithm level. In *Proceedings of the 4th International Conference on Embedded and Multimedia Computing*, pages 1–5, Jeju, Korea, 10–12 Dec. 2009. doi: 10.1109/EM-COM.2009.5402965.

[232] M. MacDiarmid and M. Bacic. Quantifying the accuracy of hardware-in-the-loop simulations. In *Proceedings of the 2007 American Control Conference*, pages 5147–5152, New York, NY, USA, 9–13 July 2007. doi: 10.1109/ACC.2007.4283062.

[233] K. Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *Proceedings of the 2nd International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, DICTAP '12, pages 177–182, Bangkok, Thailand, 16–18 May 2012. doi: 10.1109/DICTAP.2012.6215346.

[234] S. Maleki, C. Fu, A. Banotra, and Z. Zong. Understanding the impact of object oriented programming and design patterns on energy efficiency. In *Proceedings of the 8th International Green and Sustainable Computing Conference*, IGSC '17, pages 1–6, Orlando, FL, USA, 23–25 Oct. 2017. doi: 10.1109/IGCC.2017.8323605.

[235] R. Mall. *Fundamentals of Software Engineering*. Eastern Economy Edition. PHI Learning Private Limited, Delhi, India, 5th edition, 2018. ISBN 978-93-88028-02-8.

[236] D. Marculescu, R. Marculescu, and M. Pedram. Information theoretic measures of energy consumption at register transfer level. In *Proceedings of the International Symposium on Low Power Design*, ISLPED '95, pages 81–86, Dana Point, California, USA, 23–26 Apr. 1995. ISBN 978-0-89791-744-5. doi: 10.1145/224081.224096.

[237] A. Martin and M. R. Emami. Dynamic load emulation in hardware-in-the-loop simulation of robot manipulators. *IEEE Transactions on Industrial Electronics*, 58(7): 2980–2987, 2 Sept. 2011. doi: 10.1109/TIE.2010.2072890.

[238] R. C. Martin and K. Henney. *Clean Architecture: A craftsman's guide to software structure and design*. Robert C. Martin series. Prentice Hall, Boston, MA, USA, 2018. ISBN 978-0-13-449416-6.

[239] B. Martinez, M. Monton, I. Vilajosana, and J. D. Prades. The power of models: Modeling power consumption for IoT devices. *IEEE Sensors Journal*, 15(10):5777–5789, 2015. ISSN 1530-437X. doi: 10.1109/JSEN.2015.2445094.

[240] P. Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer Publishing Company, Incorporated, Cham, Switzerland, 4th edition, 2021. ISBN 978-3-030-60909-2.

[241] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, Boston, MA, USA, 2001. ISBN 978-0-201-32564-5.

[242] M. McHugh, O. Cawley, F. McCaffcry, I. Richardson, and X. Wang. An agile V-model for medical device software development to overcome the challenges with plan-driven software development lifecycles. In *Proceedings of the 5th International Workshop on Software Engineering in Health Care*, SEHC '13, pages 12–19, San Francisco, CA, USA, 20–21 May 2013. doi: 10.1109/SEHC.2013.6602471.

[243] Melexis NV. Mlx90640 32x24 ir array – datasheet, revision 12, 2021. URL https://www.melexis.com/-/media/files/documents/datasheets/mlx90640-datasheet-melexis.pdf. Last Access: August 3rd, 2022.

[244] S. J. Mellor, M. Balcer, and I. Jacoboson. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 978-0-201-74804-8.

[245] M. Menghin, N. Druml, C. Steger, h. Weiss, H. Bock, and J. Haid. Development framework for model driven architecture to accomplish power-aware embedded systems. In *Proceedings of the 17th Euromicro Conference on Digital System Design*, pages 122–128, Verona, Italy, 27–29 Aug. 2014. doi: 10.1109/DSD.2014.30.

[246] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, Mar. 2006. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.10.021. Proceedings of the International Workshop on Graph and Model Transformation.

[247] A. Meroth, F. Tränkle, B. Richter, M. Wagner, M. Neher, and J. Lüling. Optimization of the development process of intelligent transportation systems using automotive spice and iso 26262. In *Proceedings of the 17th International IEEE Conference on Intelligent Transportation Systems*, ITSC '14, pages 1481–1486, Qingdao, China, 8–11 Oct. 2014. doi: 10.1109/ITSC.2014.6957641.

[248] G. Meszaros and J. Doble. A pattern language for pattern writing. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 529–574. Addison-Wesley Longman Publishing Co., Inc., USA, 1997. ISBN 978-0-201-31011-5.

[249] A. Michailidis, U. Spieth, T. Ringler, B. Hedenetz, and S. Kowalewski. Test front loading in early stages of automotive software development based on AUTOSAR. In *Proceedings of the 2010 Design, Automation & Test in Europe Conference & Exhibition*, DATE '10, pages 435–440, Dresden, Germany, 8–10 Mar. 2010. doi: 10.1109/DATE.2010.5457166.

[250] Microchip Technology Inc. *ATWINC15x0-MR210xB IEEE 802.11 b/g/n SmartConnect IoT Module – Datasheet, Revision E*, 2021. URL https://ww1.microchip.com/downloads/en/DeviceDoc/ATWINC15x0-MR210xB-IEEE-802.11-b-g-n-SmartConnect-IoT-Module-DS70005304E.pdf. Last Access: August 3rd, 2022.

[251] Microsoft Corporation. Flowchart maker and diagramming software | microsoft visio, 2022. URL https://www.microsoft.com/en/microsoft-365/visio/flowchart-software. Last Access: August 3rd, 2022.

[252] K. Mikhaylov and J. Tervonen. Evaluation of power efficiency for digital serial interfaces of microcontrollers. In *Proceeding of the 5th International Conference on New Technologies, Mobility and Security*, NTMS '12, pages 1–5, 7–10 May 2012. doi: 10.1109/NTMS.2012.6208716.

[253] M. Miśkowicz. *Event-based control and signal processing*. Embedded systems. CRC Press, Boca Raton, FL, USA, 1st edition, 2015. ISBN 978-1-315-21507-5.

[254] Modelica Association Project DCP. DCP Specification Document, Version 1.0. Technical report, Linköping, Sweden, 2019. URL https://www.dcp-standard.org.

[255] R. Moraes, T. Basso, and E. Martins. V-model adaptation for space systems in light of the ECSS standard. In *Proceedings of the 10th Latin-American Symposium on Dependable Computing*, LADC '21, pages 1–4, Florianópolis, Brazil, 2021. doi: 10.1109/LADC53747.2021.9672593.

[256] É. Morin, M. Maman, R. Guizzetti, and A. Duda. Comparison of the device lifetime in wireless networks for the internet of things. *IEEE Access*, 5:7097–7114, 7 Apr. 2017. doi: 10.1109/ACCESS.2017.2688279.

[257] J. Morrish, M. Arnott, and M. Hatton. Global IoT forecast report, 2022-2032. Technical report, Transforma Insights, 2023. URL https://transformainsights.com/research/reports/global-iot-forecast-report-2032. Last Access: June 3rd, 2023.

[258] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, Hoboken and N.J, 3rd edition, 2012. ISBN 978-1-118-03196-4.

[259] D. P. Möller and R. E. Haas. *Guide to Automotive Connectivity and Cybersecurity: Trends, Technologies, Innovations and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2019. ISBN 978-3-319-73511-5.

[260] H. W. Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, Georg-August-Universität Göttingen, Fakultät für Mathematik und Informatik, Göttingen, Germany, 2004.

[261] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, Waltham, MA, USA, 2nd edition, 2013. ISBN 978-0-12-382196-6.

[262] Nordic Semiconductor. Online power profiler, 2022. URL https://devzone.nordicsemi.com/power/w/opp. Last Access: June 1st, 2022.

[263] A. Noureddine and A. Rajan. Optimising energy consumption of design patterns. In *Proceedings of the 37th International Conference on Software Engineering*, volume 2 of *ICSE '15*, pages 623–626, Florence Italy, 16–24 May 2015. IEEE Press. doi: 10.1109/ICSE.2015.208.

[264] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. Runtime monitoring of software energy hotspots. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, pages 160–169, Essen, Germany, 3–7 Sept. 2012. doi: 10.1145/2351676.2351699.

[265] A. Noureddine, R. Rouvoy, and L. Seinturier. Monitoring energy hotspots in software. *Automated Software Engineering*, 22(3):291–332, Sept. 2015. ISSN 0928-8910. doi: 10.1007/s10515-014-0171-1.

[266] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta. Comparison of JSON and XML data interchange formats: A case study. In D. Che, editor, *Proceedings of the 22nd International Conference on Computer Applications in Industry and Engineering (CAINE)*, pages 157–162, San Francisco, CA, USA, 4–6 Nov. 2009. ISCA.

[267] NXP Semiconductors. MPC8641 and MPC8641D integrated host processor hardware specifications. document number: MPC8641D rev. 3, 2014. URL https://www.nxp.com/docs/en/data-sheet/MPC8641DEC.pdf. Last Access: August 3rd, 2022.

[268] NXP Semiconductors. AN11783 - CLRC663 pluslow power card detection. Rev. 1.0, 2017. URL https://www.nxp.com/docs/en/application-note/AN11783.pdf. Last Access: August 3rd, 2020.

[269] NXP Semiconductors. LPC5411x product datasheet. rev. 2.6, 2020. URL https://www.nxp.com/docs/en/data-sheet/LPC5411X.pdf. Last Access: August 3rd, 2022.

[270] Object Management Group. UML Profile for Schedulability, Performance, & Time. Version 1.1. OMG document number formal/05-01-02 (https://www.omg.org/spec/SPTP/), 2005. Last Access: August 7th, 2022.

[271] Object Management Group. UML Profile for Modeling QoS and FT. Version 1.1. OMG document number formal/08-04-05 (https://www.omg.org/spec/QFTP/), 2008. Last Access: August 7th, 2022.

[272] Object Management Group. Model Driven Architecture (MDA): MDA Guide rev. 2.0. OMG document number ormsc/2014-06-01 (https://www.omg.org/cgi-bin/doc?ormsc/14-06-01), 2014. Last Access: August 7th, 2022.

[273] Object Management Group. XML Metadata Interchange. Version 2.5.1. OMG document number formal/15-06-07 (https://www.omg.org/spec/XMI), 2015. Last Access: August 7th, 2022.

[274] Object Management Group. Requirements Interchange Format (ReqIF). Version 1.2. OMG document number formal/16-07-01 (https://www.omg.org/spec/ReqIF/1.2/), 2016. Last Access: August 7th, 2022.

[275] Object Management Group. Unified Modeling Language. Version 2.5.1. OMG document number formal/17-12-05 (https://www.omg.org/spec/UML/2.5.1/), 2017. Last Access: August 7th, 2022.

[276] Object Management Group. UML Testing Profile 2 Specification. Version 2.1. OMG document number formal/20-08-05 (https://www.omg.org/spec/UTP2/2.1/), 2017. Last Access: August 7th, 2022.

[277] Object Management Group. Action Language for Foundational UML. Version 1.1. OMG document number formal/17-07-04 (https://www.omg.org/spec/ALF/1.1/), 2017. Last Access: August 7th, 2022.

[278] Object Management Group. A UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. Version 1.2. OMG document number formal/19-04-01 (https://www.omg.org/spec/MARTE/1.2/), 2019. Last Access: August 7th, 2022.

[279] Object Management Group. OMG System Modeling Language Specification. Version 1.6. OMG document number formal/19-11-01 (https://www.omg.org/spec/SysML/1.6/), 2019. Last Access: August 7th, 2022.

[280] Object Management Group. Precise Semantics of UML State Machines. Version 1.0. OMG document number formal/19-05-01 (https://www.omg.org/spec/PSSM/1.0), 2019. Last Access: August 7th, 2022.

[281] Object Management Group. Semantics of a Foundational Subset for Executable UML Models. Version 1.5. OMG document number formal/21-03-01 (https://www.omg.org/spec/FUML/1.5/), 2021. Last Access: August 7th, 2022.

[282] E. E. Ogheneovo. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 02(14):1–16, 2014. ISSN 2327-5219. doi: 10.4236/jcc.2014.214001.

[283] OpenRPC. *OpenRPC Specification Version 1.2.6*, 2020. URL https://spec.open-rpc.org/. Last Access: June 1st, 2022.

[284] F. Oquendo, J. Leite, and T. Batista. Eliciting requirements of software architectures. In F. Oquendo, J. Leite, and T. Batista, editors, *Software Architecture in Action*, Undergraduate Topics in Computer Science, pages 27–36. Springer International Publishing, Cham, Switzerland, 2016. ISBN 978-3-319-44337-9. doi: 10.1007/978-3-319-44339-3_3.

[285] R. Oshana and M. Kraeling. *Software engineering for embedded systems: Methods, practical techniques, and applications.* Newnes, Kidlington, Oxfordshire, UK, 2nd edition, 2019. ISBN 978-0-12-809448-8.

[286] Osnabrück University. Holistic model-driven development for embedded systems in consideration of diverse hardware architectures., 2022. URL https://www.informatik.uni-osnabrueck.de/arbeitsgruppen/software_engineering/research/holmes.html. Last Access: August 3rd, 2022.

[287] S. Ould and N. S. Bennett. Energy performance analysis and modelling of lora prototyping boards. *Sensors*, 21(23), 2021. ISSN 1424-8220. doi: 10.3390/s21237992.

[288] M. Ozkaya. Are the UML modelling tools powerful enough for practitioners? a literature review. *IET Software*, 13(5):338–354, 2019. doi: 10.1049/iet-sen.2018.5409.

[289] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016. ISSN 0740-7459. doi: 10.1109/MS.2015.83.

[290] M. Panunzio and T. Vardanega. An architectural approach with separation of concerns to address extra-functional requirements in the development of embedded real-time software systems. *Journal of Systems Architecture*, 60(9):770–781, 2014. ISSN 13837621. doi: 10.1016/j.sysarc.2014.06.001.

[291] L. Papadopoulos, C. Marantos, G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou, and D. Soudris. Interrelations between software quality metrics, performance and energy consumption in embedded applications. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, pages 62–65, Sankt Goar, Germany, 28–30 May 2018. ISBN 978-1-4503-5780-7. doi: 10.1145/3207719.3207736.

[292] Ó. Pastor and J. C. Molina. *Model-driven architecture in practice: A software production environment based on conceptual modeling.* Springer, Berlin, Germany, 2007. ISBN 978-3-540-71867-3.

[293] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X '11, Cambridge, MA, USA, 14–15 Nov. 2011. ACM. ISBN 978-1-4503-1059-8. doi: 10.1145/2070562.2070567.

[294] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with Eprof. In *7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, Bern, Switzerland, 10–13 Apr. 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168841.

[295] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, Low Wood Bay, Lake District, UK, 25–26 June 2012. ACM. ISBN 978-1-4503-1301-8. doi: 10.1145/2307636.2307661.

[296] P. S. Patil, J. Doshi, and D. Ambawade. Reducing power consumption of smart device by proper management of wakelocks. In *Proceedings of the 2015 IEEE International Advance Computing Conference*, IACC '15, pages 883–887, Banglore, India, 12–13 June 2015. doi: 10.1109/IADCC.2015.7154832.

[297] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. The Morgan Kaufmann series in computer architecture and design. Morgan Kaufmann, San Francisco, USA, 6th edition, 2020. ISBN 978-0-12-820109-1.

[298] J. K. Peckol. *Embedded Systems: A Contemporary Design Tool*. John Wiley & Sons Ltd, Hoboken, NJ, USA, 2nd edition, 2019. ISBN 978-1-119-45750-3.

[299] B. Peischl, M. Weiglhofer, and F. Wotawa. Executing abstract test cases. In R. Koschke, O. Herzog, K. Rödiger, and M. Ronthaler, editors, *37. Jahrestagung der Gesellschaft für Informatik, Informatik trifft Logistik*, volume P-110 of *LNI*, pages 416–421, Bremen, Germany, 24–27 Sept. 2007. GI.

[300] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, ISLPED '98, pages 76–81, Monterey, CA, USA, 10–12 Aug. 1998. doi: 10.1145/280756.280790.

[301] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 22–31, Hyderabad, India, 31 May–1 June 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597110.

[302] Plantower Technology Co., Ltd. *PMS1003 – Datasheet, Version 1.0*, 2021. URL https://plantower.com/static/upload/file/20220627/1656292073878896.pdf. Last Access: March 3rd, 2023.

[303] K. Pohl and C. Rupp. *Requirements engineering fundamentals: A study guide for the certified professional for requirements engineering exam, foundation level, IREB compliant*. Rocky Nook, Santa Barbara, CA, USA, 2nd edition, 2015. ISBN 978-1-937538-77-4.

[304] J. Porter, G. Karsai, P. Völgyesi, H. Nine, P. Humke, G. Hemingway, R. Thibodeaux, and J. Sztipanovits. Towards model-based integration of tools and techniques for embedded

control system design, verification, and implementation. In M. R. V. Chaudron, editor, *Models in Software Engineering: Workshops and Symposia at MODELS 2008. Reports and Revised Selected Papers*, MODELS '08, pages 20–34, Toulouse, France, 28 Sept.–3 Oct. 2009. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-01648-6_3.

[305] J. Porter, P. Volgyesi, N. Kottenstette, H. Nine, G. Karsai, and J. Sztipanovits. An experimental model-based rapid prototyping environment for high-confidence embedded software. In *2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 3–10, 23–26 June 2009. doi: 10.1109/RSP.2009.32.

[306] W. Prenninger and A. Pretschner. Abstractions for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:59–71, 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2004.02.086. Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004).

[307] G. Procaccianti, S. Bevini, and P. Lago. Energy efficiency in cloud software architectures. In B. Page, A. G. Fleischer, J. Göbel, and V. Wohlgemuth, editors, *Proceedings of the 27th International Conference on Environmental Informatics for Environmental Protection, Sustainable Development and Risk Management*, EnviroInfo '13, pages 291–299, Hamburg, Germany, 2–4 Sept. 2013. Shaker.

[308] G. Procaccianti, P. Lago, and G. A. Lewis. A catalogue of green architectural tactics for the cloud. In *Proceedings of the 8th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 29–36, Victoria, BC, Canada, 29 Sept. 2014. doi: 10.1109/MESOCA.2014.12.

[309] G. Procaccianti, P. Lago, and G. A. Lewis. Green architectural tactics for the cloud. In *Proceedings of the 2014 IEEE/IFIP Conference on Software Architecture*, WICSA '14, pages 41–44, Sydney, Australia, 7–11 Apr. 2014. doi: 10.1109/WICSA.2014.30.

[310] D. P. Proos and N. Carlsson. Performance comparison of messaging protocols and serialization formats for digital twins in IoV. In *2020 IFIP Networking Conference (Networking)*, pages 10–18, Paris, France, 22–26 June 2020.

[311] S. J. Prowell. Using Markov chain usage models to test complex systems. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, volume 9 of *HICSS '05*, page 318c, Big Island, HI, USA, 3–6 Jan. 2005. ISBN 978-0-7695-2268-5. doi: 10.1109/HICSS.2005.663.

[312] Qoitech AB. The otii server, 2021. URL https://www.qoitech.com/help/tcpserver/. Last Access: August 3rd, 2022.

[313] Qoitech AB. Otii arc, 2023. URL https://www.qoitech.com/otii-arc-pro/. Last Access: July 3rd, 2023.

[314] Qt Group. Qt 5.15, 2022. URL https://doc.qt.io/qt-5/. Last Access: August 3rd, 2022.

[315] G. Qu, N. Kawabe, K. Usami, and M. Potkonjak. Function-level power estimation methodology for microprocessors. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 810–813, Los Angeles, CA, USA, 5–9 June 2000. ACM. ISBN 978-1-58113-187-1. doi: 10.1145/337292.337786.

[316] A. Raghunathan, S. Dey, and N. Jha. Register-transfer level estimation techniques for switching activity and power consumption. In *Proceedings of the International Conference on Computer Aided Design*, pages 158–165, San Jose, CA, USA, 10–14 Nov. 1996. doi: 10.1109/ICCAD.1996.569539.

[317] RAKwireless Technology Co. *RAK811-Module: Datasheet*, 2021. URL https://docs.rakwireless.com/Product-Categories/WisDuo/RAK811-Module/Datasheet/. Last Access: August 3rd, 2022.

[318] Raspberry Pi Ltd. Raspberry Pi Zero W, 2023. URL https://www.raspberrypi.com/products/raspberry-pi-zero-w/. Last Access: April 1st, 2023.

[319] A. Ray, C. Ackermann, R. Cleaveland, C. Shelton, and C. Martin. Functional and nonfunctional design verification for embedded software systems. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 83, chapter 6, pages 277–321. Elsevier, 2011. doi: 10.1016/B978-0-12-385510-7.00006-0.

[320] G. Reggio, M. Leotta, F. Ricca, and D. Clerissi. What are the used UML diagrams? A preliminary survey. In *Proceedings of 3rd International Workshop on Experiences and Empirical Studies in Software Modelling*, volume 1078 of *EESSMod '13*, pages 3–12, 2013.

[321] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg. Internet of things patterns. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, EuroPlop '16, Kaufbeuren, Germany, 6–10 July 2016. ACM. ISBN 978-1-4503-4074-8. doi: 10.1145/3011784.3011789.

[322] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg. Internet of things patterns for devices. In *Proceedings of the 9th International Conferences on Pervasive Patterns and Applications*, PATTERNS '17, pages 117–126, Athens, Greece, 19–23 Feb. 2017.

[323] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg. Internet of things patterns for devices: Powering, operating, and sensing. *International Journal on Advances in Internet Technology*, 10(3&4):106–123, 2017.

[324] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg. Internet of things patterns for device bootstrapping and registration. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, 12–16 July 2017. doi: 10.1145/3147704.3147721.

[325] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg. Internet of things patterns for communication and management. In J. Noble, R. Johnson, U. Zdun, and E. Wallingford, editors, *Transactions on Pattern Languages of Programming IV*, volume 10600 of *Lecture Notes in Computer Science*, pages 139–182. Springer International Publishing, Cham, Switzerland, 2019. ISBN 978-3-030-14290-2. doi: 10.1007/978-3-030-14291-9_5.

[326] A. Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015. ISSN 1477-8424. doi: https://doi.org/10.1016/j.cl.2015.06.001.

[327] F. T. Rodríguez, F. Lonetti, A. Bertolino, M. P. Usaola, and B. P. Lamancha. Extending UML testing profile towards non-functional test modeling. In *Proceeding of the 2nd International Conference on Model-Driven Engineering and Software Development*, MODELSWARD '14, pages 488–497, Lisbon, Portugal, 7–9 Jan. 2014.

[328] D. Rossi, I. Loi, A. Pullini, and L. Benini. Ultra-low-power digital architectures for the internet of things. In *Enabling the Internet of Things: From Integrated Circuits to Integrated Systems*, volume 59, pages 69–93. Springer International Publishing, Cham, Switzerland, 2017. ISBN 978-3-319-51480-2. doi: 10.1007/978-3-319-51482-6_3.

[329] A. Sabbaghi and M. R. Keyvanpour. State-based models in model-based testing: A systematic review. In *Proccedings of the 4th International Conference on Knowledge-Based Engineering and Innovation*, KBEI '17, pages 942–948, Tehran, Iran, 22 Dec. 2017. doi: 10.1109/KBEI.2017.8324934.

[330] S. A. Safdar, M. Z. Iqbal, and M. U. Khan. Empirical evaluation of UML modeling tools – a controlled experiment. In G. Taentzer and F. Bordeleau, editors, *Modelling Foundations and Applications*, volume 9153 of *Lecture Notes in Computer Science*, pages 33–44. Springer International Publishing, Cham, Switzerland, 2015. ISBN 978-3-319-21150-3. doi: 10.1007/978-3-319-21151-0_3.

[331] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. In *Proceedings of the 1st International Workshop on Green and Sustainable Software*, GREENS '12, pages 55–61, Zurich, Switzerland, 3 June 2012. doi: 10.1109/GREENS.2012.6224257.

[332] K. A. Saleh. *Software Engineering*. J. Ross Publishing, Inc., Fort Lauderdale, FL, USA, 2009. ISBN 978-1-932159-94-3.

[333] K. O. M. Salih, T. A. Rashid, D. Radovanovic, and N. Bacanin. A comprehensive survey on the internet of things with the industrial marketplace. *Sensors*, 22(3), 19 Jan. 2022. ISSN 1424-8220. doi: 10.3390/s22030730.

[334] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstić, J. M. Lourenço, D. Nickovic, G. J. Pace, J. Rufino, J. Signoles, D. Traytel, and A. Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods in System Design*, 54(3): 279–335, 2019. ISSN 1572-8102. doi: 10.1007/s10703-019-00337-w.

[335] D. Sas and P. Avgeriou. Quality attribute trade-offs in the embedded systems industry: an exploratory case study. *Software Quality Journal*, 28(2):505–534, 2020. ISSN 1573-1367. doi: 10.1007/s11219-019-09478-x.

[336] M. Schaarschmidt, C. Fuhrmann, M. Uelschen, C. Westerkamp, and E. Pulvermüller. Energieeffiziente Entwurfsmuster für das Internet der Dinge - Möglichkeiten und Perspektiven für Single- und Multicore. In *Tagungsband Embedded Software Engineering Kongress*, pages 511–522, Sindelfingen, Germany, 3–7 Dec. 2018. (in German).

[337] M. Schaarschmidt, M. Uelschen, E. Pulvermüller, and C. Westerkamp. Framework of software design patterns for energy-aware embedded systems. In *Proceedings of the 15th*

International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE '20, pages 62–73, Online Streaming, 5–6 May 2020. INSTICC, SciTePress. ISBN 978-989-758-421-3. doi: 10.5220/0009351000620073.

[338] M. Schaarschmidt, M. Uelschen, E. Pulvermüller, and C. Westerkamp. Energy-aware pattern framework: The energy-efficiency challenge for embedded systems from a software design perspective. In R. Ali, H. Kaindl, and L. A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, volume 1375 of *Communications in Computer and Information Science*, pages 182–207, Cham, Switzerland, 27 Feb. 2021. Springer International Publishing. ISBN 978-3-030-70006-5. doi: 10.1007/978-3-030-70006-5_8.

[339] M. Schaarschmidt, M. Uelschen, and E. Pulvermüller. Power consumption estimation in model driven software development for embedded systems. In *Proceedings of the 16th International Conference on Software Technologies*, volume 1 of *ICSOFT '21*, pages 47–58, Online Streaming, 6–8 July 2021. INSTICC, SciTePress. ISBN 978-989-758-523-4. doi: 10.5220/0010522700470058.

[340] M. Schaarschmidt, M. Uelschen, and E. Pulvermüller. Towards power consumption optimization for embedded systems from a model-driven software development perspective. In H.-G. Fill, M. van Sinderen, and L. A. Maciaszek, editors, *International Conference on Software Technologies*, volume 1622 of *Communications in Computer and Information Science*, pages 117–142, Cham, Switzerland, 18 July 2022. Springer International Publishing. ISBN 978-3-031-11513-4. doi: 10.1007/978-3-031-11513-4_6.

[341] M. Schaarschmidt, M. Uelschen, and E. Pulvermüller. Hunting energy bugs in embedded systems: A software-model-in-the-loop approach. *Electronics*, 11(13), 2022. ISSN 2079-9292. doi: 10.3390/electronics11131937. URL https://www.mdpi.com/2079-9292/11/13/1937.

[342] P. R. Schaumont. *A practical introduction to hardware/software codesign*. Springer Science & Business Media, New York, NY, USA, 2012. ISBN 978-1-46143736-9. doi: 10.1007/978-1-4614-3737-6.

[343] I. Schieferdecker and A. Hoffmann. Model-based testing. In P. A. Laplante, editor, *Encyclopedia of Software Engineering*, pages 556–570. Taylor & Francis, London, UK, 2010. doi: 10.1081/E-ESE-120044686.

[344] M. Schneider, H. Blume, and T. G. Noll. Power estimation on functional level for programmable processors. *Advances in Radio Science*, 2:215–219, 2004. doi: 10.5194/ars-2-215-2004.

[345] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons Ltd, Chichester, UK, 2005. ISBN 978-0-470-85884-4.

[346] M. Segura, T. Poggi, and R. Barcena. A generic interface for x-in-the-loop simulations based on distributed co-simulation protocol. *IEEE Access*, 11:5578–5595, 2023. doi: 10.1109/ACCESS.2023.3237075.

[347] M. Seidl, M. Scholz, C. Huemer, and G. Kappel. *UML @ classroom: An introduction to object-oriented modeling*. Undergraduate Topics in Computer Science. Springer, Cham, Switzerland, 1st edition, 2015. ISBN 978-3-31912741-5.

[348] B. Selic and S. Gérard. *Modeling and analysis of real-time and embedded systems with UML and MARTE: Developing cyber-physical systems*. Morgan Kaufmann, Waltham, MA, 2014. ISBN 978-0-124-16656-1.

[349] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 1994. ISBN 978-0-471-59917-3.

[350] Semtech Corporation. *An In-depth Look at LoRaWAN™ Class A Devices*, 2019. URL https://lora-developers.semtech.com/uploads/documents/files/LoRaWAN_Class_A_Devices_In_Depth_Downloadable.pdf. Last Access: August 3rd, 2022.

[351] Semtech Corporation. *SX1276/77/78/79 Datasheet Rev. 7*, 2020. URL https://www.semtech.com/products/wireless-rf/lora-core/sx1276. Last Access: August 3rd, 2022.

[352] E. Senn, N. Julien, J. Laurent, and E. Martin. Power consumption estimation of a C program for data-intensive applications. In G. Goos, J. Hartmanis, J. van Leeuwen, B. Hochet, A. J. Acosta, and M. J. Bellido, editors, *Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation*, volume 2451 of *Lecture Notes in Computer Science*, pages 332–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-44143-4. doi: 10.1007/3-540-45716-X_33.

[353] E. Senn, J. Laurent, E. Juin, and J.-P. Diguet. Refining power consumption estimations in the component based AADL design flow. In *Proceedings of the 2008 Forum on Specification, Verification and Design Languages*, pages 173–178, Stuttgart, Germany, 23–25 Sept. 2008. IEEE. doi: 10.1109/FDL.2008.4641441.

[354] N. Sharma, M. Shamkuwar, and I. Singh. The history, present and future with IoT. In V. E. Balas, V. K. Solanki, R. Kumar, and M. Khari, editors, *Internet of Things and Big Data Analytics for Smart Generation*, pages 27–51. Springer International Publishing, Cham, Switzerland, 2019. ISBN 978-3-030-04203-5. doi: 10.1007/978-3-030-04203-5_3.

[355] H. Shokry and M. Hinchey. Model-based verification of embedded software. *IEEE Computer*, 42:53–59, 04 2009. doi: 10.1109/MC.2009.125.

[356] T. Shu, M. Xia, J. Chen, and C. De Silva. An energy efficient adaptive sampling algorithm in a sensor network for automated water quality monitoring. *Sensors*, 17(11), 2017. ISSN 1424-8220. doi: 10.3390/s17112551. URL https://www.mdpi.com/1424-8220/17/11/2551.

[357] S. Siegl., V. Entin., R. German., and G. Kiffe. Model driven testing with time augmented Markov chain usage - computations and test case generation algorithms for time augmented markov chain usage models. In *Proceedings of the 4th International Conference on Software and Data Technologies*, volume 1 of *ICSOFT '09*, pages 202–207. INSTICC, SciTePress, 2009. ISBN 978-989-674009-2. doi: 10.5220/0002255902020207.

[358] S. Siegl, K.-S. Hielscher, and R. German. Introduction of time dependencies in usage model based testing of complex systems. In *Proceedings of the 2010 IEEE*

International Systems Conference, pages 622–627, San Diego, CA, USA, 5–8 Apr. 2010. doi: 10.1109/SYSTEMS.2010.5482341.

[359] Silicon Laboratories. *CP2102/9 single-chip usb-to-uart bridge*, 2017. URL https://www.silabs.com/documents/public/data-sheets/CP2102-9.pdf. Last Access: August 3rd, 2022.

[360] J. Singh, K. Naik, and V. Mahinthan. Impact of developer choices on energy consumption of software on servers. *Procedia Computer Science*, 62:385–394, 2015. ISSN 18770509. doi: 10.1016/j.procs.2015.08.423.

[361] C. U. Smith and L. G. Williams. *Software Performance Engineering*, pages 343–365. Springer US, Boston, MA, USA, 2003. ISBN 978-0-306-48738-5. doi: 10.1007/0-306-48738-1_16.

[362] I. Sommerville. *Software engineering*. Pearson, Harlow, Essex, England, 10th edition, 2016. ISBN 978-1-292-09613-1.

[363] M. Sourouri, E. B. Raknes, N. Reissmann, J. Langguth, D. Hackenberg, R. Schöne, and P. G. Kjeldsberg. Towards fine-grained dynamic tuning of hpc applications on modern multi-core architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, Denver, CO, USA, 12–17 Nov. 2017. doi: 10.1145/3126908.3126945.

[364] SparxSystems Software GmbH. Enterprise architect, 2022. URL https://www.sparxsystems.eu/enterprise-architect/. Last Access: August 3rd, 2022.

[365] A. Speck. Robot simulation and monitoring on real controllers (RoboSiM). In *10th European Simulation Symposium and Exhibition*, ESS '98, pages 482–489, Nottingham, UK, 26–28 Oct. 1998.

[366] A. Speck and H. Klaeren. Robosim: Java 3D robot visualization. In *25th Annual Conference of the IEEE Industrial Electronics Society*, volume 2 of *IECON '99*, pages 821–826, San Jose, CA, USA, 29 Nov.–3 Dec. 1999. doi: 10.1109/IECON.1999.816506.

[367] A. Spillner and T. Linz. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. dpunkt, Heidelberg, Germany, 5th edition, 2021. ISBN 978-3-86490-834-7.

[368] A. Spillner and K. Vosseberg. The W-MODEL. strengthening the bond between development and test. In *Proceedings of the Software Testing Analysis & Review Conference*, STAReast '02, pages 15–17, Orlando, FL, USA, 13–17 May 2002.

[369] T. Stahl and M. Völter. *Model-driven software development: Technology, engineering, management*. John Wiley, Chichester, UK, 2006. ISBN 978-0-470-02570-3.

[370] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proceedings of the International Workshop on Power And Timing Modeling, Optimization and Simulation*, PATMOS '01, Yverdon-les-Bains, Switzerland, 26–28 Sept. 2001.

[371] STMicroelectronics. *STM32L476xx Datasheet*, 2019. URL https://www.st.com/resource/en/datasheet/stm32l476je.pdf. Last Access: August 3rd, 2022.

[372] STMicroelectronics. STM32CubeMX, 2022. URL https://www.st.com/en/development-tools/stm32cubemx.html. Last Access: June 1st, 2022.

[373] A. Sumaray and S. K. Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, Kuala Lumpur, Malaysia, 20–22 Feb. 2012. ACM. ISBN 978-1-4503-1172-4. doi: 10.1145/2184751.2184810.

[374] J. Svennebring, J. Logan, J. Engblom, and P. Strömblad. Embedded multicore: An introduction (rev. 0), 2009. URL https://www.nxp.com/files-static/32bit/doc/ref_manual/EMBMCRM.pdf. Last Access: August 3rd, 2020.

[375] T. K. Tan, A. Raghunathan, and N. K. Jha. Software architectural transformations: a new approach to low energy embedded software. In *Design, Automation, and Test in Europe Conference and Exhibition*, pages 1046–1051, Munich, Germany, 7 Mar. 2003. IEEE Computer Society. ISBN 978-0-7695-1870-1. doi: 10.1109/DATE.2003.1253742.

[376] K. Tanaka, S. Inaho, M. Hatano, Y. Kuroki, M. Yoo, and T. Yokoyama. An extended Simulink to UML model transformation tool for embedded control software development. In *Proceedings of the 2017 International Conference on Industrial Design Engineering*, ICIDE '2017, pages 76–81, Dubai, United Arab Emirates, 29–31 Dec. 2017. doi: 10.1145/3178264.3178284.

[377] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier. Formalizing execution semantics of UML profiles with fUML models. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 133–148. Springer International Publishing, Cham, Switzerland, 2014. ISBN 978-3-319-11652-5. doi: 10.1007/978-3-319-11653-2_9.

[378] Texas Instruments Incorporated. Ultra-low power (ULP) advisor, 2022. URL https://software-dl.ti.com/ccs/esd/documents/ccs_ulp_advisor.html. Last Access: August 3rd, 2022.

[379] Texas Instruments Incorporated. MSP430 microcontrollers, 2022. URL https://www.ti.com/microcontrollers-mcus-processors/microcontrollers/msp430-microcontrollers/overview.html. Last Access: August 3rd, 2022.

[380] The MathWorks, Inc. MATLAB, 2021. URL https://www.mathworks.com/products/matlab. Last Access: August 3rd, 2022.

[381] The MathWorks, Inc. ThingSpeak, 2021. URL https://www.thingspeak.com/. Last Access: August 3rd, 2022.

[382] The MathWorks, Inc. Arduino Support – MATLAB & Simulink, 2022. URL https://www.mathworks.com/products/hardware/arduino.html. Last Access: August 3rd, 2022.

[383] The MathWorks, Inc. Simulink - Simulation and Model-Based Design, 2022. URL https://www.mathworks.com/products/simulink.html. Last Access: August 3rd, 2022.

[384] The MathWorks, Inc. Stateflow, 2022. URL https://www.mathworks.com/products/stateflow.html. Last Access: August 3rd, 2022.

[385] The Things Industries B.V. The things network, 2022. URL https://www.thethingsnetwork.org/. Last Access: August 3rd, 2022.

[386] J. Tidwell, C. Brewer, and A. Valencia. *Designing interfaces*. O'Reilly Media, Inc., Sebastopol, CA, USA, 3rd edition, 2020. ISBN 978-1-4920-5196-1.

[387] V. Tiwari, S. Malik, A. Wolfe, and M.-C. Lee. Instruction level power analysis and optimization of software. In *Proceedings of the 9th International Conference on VLSI Design*, pages 326–328, Bangalore, India, 3–6 Jan. 1996. doi: 10.1109/ICVD.1996.489624.

[388] A. Tobola, F. J. Streit, C. Espig, O. Korpok, C. Sauter, N. Lang, B. Schmitz, C. Hofmann, M. Struck, C. Weigand, H. Leutheuser, B. M. Eskofier, and G. Fischer. Sampling rate impact on energy consumption of biomedical signal processing systems. In *Proceedings of the 2015 IEEE 12th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*, pages 1–6, Cambridge, MA, USA, 9–12 June 2015. doi: 10.1109/BSN.2015.7299392.

[389] C. Trabelsi, R. Ben Atitallah, S. Meftali, J.-L. Dekeyser, and A. Jemai. A model-driven approach for hybrid power estimation in embedded systems design. *EURASIP Journal on Embedded Systems*, 2011(1), 2011. doi: 10.1155/2011/569031.

[390] F. Tränkle. *Modellbasierte Entwicklung Mechatronischer Systeme: mit Software- und Simulationsbeispielen für Autonomes Fahren*. De Gruyter Oldenbourg, 2021. ISBN 978-3-11072352-6. doi: 10.1515/9783110723526. (in German).

[391] M. Uelschen and M. Schaarschmidt. Software design of energy-aware peripheral control for sustainable internet-of-things devices. In *Proceedings of the 55th Hawaii International Conference on System Sciences*, HICSS '22, Maui, HI, USA, 4–7 Jan. 2022. doi: 10.24251/HICSS.2022.933.

[392] M. Uelschen, M. Schaarschmidt, C. Fuhrmann, and C. Westerkamp. PowerMonitor: Design pattern for modelling energy-aware embedded systems: Work-in-progress. In *Proceedings of the International Conference on Embedded Software Companion*, EMSOFT '19, New York, NY, USA, 13–18 Oct. 2019. ISBN 978-1-4503-6924-4. doi: 10.1145/3349568.3351551.

[393] M. Uelschen, M. Schaarschmidt, and J. Budde. Rapid-prototyping and early validation of software models through uniform integration of hardware. In *Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems*, MODELS '23, Västerås, Sweden, 1–6 Oct. 2023. doi: 10.1109/MODELS58315.2023.00019.

[394] B. Unhelkar. *Software Engineering with UML*. CRC Press, Boca Raton, FL, USA, 1st edition, 2017. ISBN 978-1-138-29743-2. doi: doi.org/10.1201/9781351235181.

[395] F. Ünlü, L. Wawrla, and A. Diaz. Energy harvesting technologies for iot edge devices. *Energy Efficient End-use Equipment International Energy Agency*, page 70, July 2018. URL https://www.iea-4e.org/wp-content/uploads/publications/2018/07/Energy_Harvesting_Final_Report.pdf. Last Access: August 3rd, 2022.

[396] P. Urard and M. Vučinić. IoT nodes: System-level view. In *Enabling the Internet of Things*, volume 29, pages 47–68. Springer International Publishing, Cham, Switzerland, 2017. ISBN 978-3-319-51480-2. doi: 10.1007/978-3-319-51482-6_2.

[397] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1st edition, 2006. ISBN 978-0-12-372501-1.

[398] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. In *Working paper series*, pages 1–18. The University of Waikato, Department of Computer Science, Hamilton, New Zealand, 2006.

[399] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification & Reliability*, 22(5):297–312, Aug. 2012. ISSN 0960-0833. doi: 10.1002/stvr.456.

[400] L. S. Vailshery. Number of internet of things (IoT) connected devices worldwide from 2019 to 2030, by communications technology. Technical report, Statista GmbH, Statista, 2022. URL https://www.statista.com/statistics/1194688/iot-connected-devices-communications-technology/. Last Access: December 6th, 2022.

[401] L. S. Vailshery. Number of internet of things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030. Technical report, Statista GmbH, Statista, 2022. URL https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. Last Access: December 6th, 2022.

[402] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, pages 429–528. Springer Berlin Heidelberg, Berlin/Heidelberg, Germany, 1998. ISBN 978-3-540-49442-3. doi: 10.1007/3-540-65306-6_21.

[403] D. Vandevoorde, N. M. Josuttis, and D. Gregor. *C++ templates: The complete guide*. Addison-Wesley, Boston, MA, USA, 2nd edition, 2018. ISBN 978-0-321-71412-1.

[404] VDI/VDE. Vdi/vde 2206: Development of mechatronic and cyber-physical systems. Technical report, VDI, Düsseldorf, Germany, 2021.

[405] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. *ACM SIGARCH Computer Architecture News*, 38(1):205–218, 13 Mar. 2010. ISSN 0163-5964. doi: 10.1145/1735970.1736044.

[406] A. Verma, A. Khatana, and S. Chaudhary. A comparative study of black box testing and white box testing. *International Journal of Computer Sciences and Engineering (JSCSE)*, 5:301–304, Dec. 2017. ISSN 2347-2693. doi: 10.26438/ijcse/v5i12.301304.

[407] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Proceedings of the 2009 Design, Automation & Test in Europe Conference & Exhibition*, DATE '09, pages 226–231, Nice, France, 20–24 Apr. 2009. doi: 10.1109/DATE.2009.5090662.

[408] Visual Paradigm International. Visual paradigm, 2022. URL https://www.visual-paradigm.com/. Last Access: August 3rd, 2022.

[409] M. C. Vuran, A. Salam, R. Wong, and S. Irmak. Internet of underground things in precision agriculture: Architecture and technology aspects. *Ad Hoc Networks*, 81: 160–173, 2018. ISSN 1570-8705. doi: 10.1016/j.adhoc.2018.07.017.

[410] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines*. Taylor & Francis, London, UK, 1st edition, 2006. ISBN 978-0-429-12137-1. doi: 10.1201/9781420013641.

[411] M. Wagner, A. Meroth, and D. Zöbel. Developing self-adaptive automotive systems. *Design Automation for Embedded Systems*, 18(3–4):199–221, Sept. 2014. ISSN 0929-5585. doi: 10.1007/s10617-013-9124-3.

[412] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1(5):172–179, Oct. 2007. ISSN 1751-8806. doi: 10.1049/iet-sen:20060076.

[413] R. Wei, D. S. Kolovos, A. Garcia-Dominguez, K. Barmpis, and R. F. Paige. Partial loading of XMI models. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 329–339, 2–7 Oct. 2016. doi: 10.1145/2976767.2976787.

[414] T. Wellhausen and A. Fiesser. How to write a pattern? In P. Avgeriou and A. Fiesser, editors, *Proceedings of the 16th European Conference on Pattern Languages of Programs*, EuroPLoP '11, pages 1–9, Irsee, Germany, 13–17 July 2011. doi: 10.1145/2396716.2396721.

[415] C. Westerkamp, A. Grunwald, and M. Schaarschmidt. LoRaWAN, NB IoT and other radio networks for agricultural applications. In P. Roer, editor, *ITG-Fb. 304: Mobilkommunikation – Technologien und Anwendungen*, ITG-Fachbericht, pages 127–130. VDE-Verlag, Osnabrück, Germany, 18–19 May 2022. ISBN 978-3-80075873-9.

[416] World Bank Group. Commodity markets outlook: Causes and consequences of metal price shocks. Technical report, World Bank, 2021. URL https://openknowledge.worldbank.org/handle/10986/35458. Last Access: August 3rd, 2022.

[417] A. Wright, H. Andrews, B. Hutton, and G. Dennis. JSON schema: A media type for describing JSON documents. Technical report, Internet Engineering Task Force, 2020. URL https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-01. Last Access: June 1st, 2022.

[418] C. Xiao, L. Zhao, T. Asada, W. Odendaal, and J. van Wyk. An overview of integratable current sensor technologies, 12–16 Oct. 2003.

[419] Z. Yong and Z. Haoxin. *PMSA003I Series Data Manual Version 2.6*, 2018. Last Access: August 3rd, 2022.

[420] K. Yu, D. Han, C. Youn, S. Hwang, and J. Lee. Power-aware task scheduling for big.LITTLE mobile processor. In *Proceedings of the 2013 International SoC Design Conference (ISOCC)*, pages 208–212, Busan, Korea, 17–19 Nov. 2013. IEEE. ISBN 978-1-4799-1142-4. doi: 10.1109/ISOCC.2013.6864009.

[421] J. Zander, I. Schieferdecker, and P. J. Mosterman. A taxonomy of model-based testing for embedded systems from multiple industry domains. In J. Zander, I. Schieferdecker, and P. J. Mosterman, editors, *Model-Based Testing for Embedded Systems*, chapter 1, pages 3–17. CRC Press, Boca Raton, FL, USA, Dec. 2011. ISBN 978-1-4398-1845-9.

[422] J. Zander-Nowicka. *Model-based testing of real-time embedded systems in the automotive domain*. PhD thesis, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Berlin, Germany, 2009.

[423] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014. doi: 10.1109/JIOT.2014.2306328.

[424] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '10, pages 105–114, Scottsdale, AZ, USA, 24–29 Oct. 2010.

[425] W. Zhang, J. Williamson, and L. Shang. Power dissipation. In S. Bhunia and S. Mukhopadhyay, editors, *Low-Power Variation-Tolerant Design in Nanometer Silicon*, chapter 2, pages 41–80. Springer, Boston, MA, USA, 2011. ISBN 978-1-4419-7418-1. doi: 10.1007/978-1-4419-7418-1_2.

[426] H.-Y. Zhou, D.-Y. Luo, Y. Gao, and D.-C. Zuo. Modeling of node energy consumption for wireless sensor networks. *Wireless Sensor Network*, 03(01):18–23, 2011. ISSN 1945-3078. doi: 10.4236/wsn.2011.31003.

[427] Z. Zhu, S. Olutunde Oyadiji, and H. He. Energy awareness workflow model for wireless sensor nodes. *Wireless Communications and Mobile Computing*, 14(17):1583–1600, Dec. 2014. doi: 10.1002/wcm.2302.

[428] S. Ziegler, R. C. Woodward, H. H.-C. Iu, and L. J. Borle. Current sensing techniques: A review. *IEEE Sensors Journal*, 9(4):354–376, 4 Mar. 2009. doi: 10.1109/JSEN.2009.2013914.

[429] R. Zurawski. *Embedded Systems Handbook: Networked Embedded Systems*. CRC Press, Boca Raton, FL, USA, 2017. ISBN 978-1-4398-0762-0. doi: 10.1201/9781439807620.

Publications

Parts of this work have already been published. The respective core concepts and ideas have been adapted within this thesis to provide an integrated approach. In the case of co-authored publications, the author of this work has contributed fundamental and conceptual ideas to the jointly and equally developed concepts. The author has also supervised student theses that provided implementation contributions to presented concepts or adapted presented concepts to address research questions not directly related to the core topics of this thesis. The publications which are part of this thesis are listed below by their publication date in descending order.

Journals

- M. Schaarschmidt, M. Uelschen, and E. Pulvermüller. Hunting energy bugs in embedded systems: A software-model-in-the-loop approach. *Electronics*, 11(13), 2022. ISSN 2079-9292. doi: 10.3390/electronics11131937. URL https://www.mdpi.com/2079-9292/11/13/

Conferences (Peer-Reviewed)

- M. Uelschen, M. Schaarschmidt, and J. Budde. Rapid-prototyping and early validation of software models through uniform integration of hardware. In *Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems*, MODELS '23, Västerås, Sweden, 1–6 Oct. 2023. doi: 10.1109/MODELS58315.2023.00019.

- M. Uelschen and M. Schaarschmidt. Software design of energy-aware peripheral control for sustainable internet-of-things devices. *In Proceedings of the 55th Hawaii International Conference on System Sciences*, HICSS '22, Maui, HI, USA, 4–7 Jan. 2022. doi: 10.24251/HICSS.2022.933

- M. Schaarschmidt, M. Uelschen, and E. Pulvermüller. Power consumption estimation in model driven software development for embedded systems. *In Proceedings of the 16th International Conference on Software Technologies*, volume 1 of ICSOFT '21, pages 47–58, Online Streaming, 6–8 July 2021. INSTICC, SciTePress. ISBN 978-989-758-523-4. doi: 10.5220/0010522700470058

- M. Schaarschmidt, M. Uelschen, E. Pulvermüller, and C. Westerkamp. Framework of software design patterns for energy-aware embedded systems. *In Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*, ENASE '20, pages 62–73, Online Streaming, 5–6 May 2020. INSTICC, SciTePress. ISBN 978-989-758-421-3. doi: 10.5220/0009351000620073. **Best Student Paper Award.**

- M. Uelschen, M. Schaarschmidt, C. Fuhrmann, and C. Westerkamp. PowerMonitor: Design pattern for modeling energy-aware embedded systems: Work-in-progress. *In Proceedings of the International Conference on Embedded Software Companion*, EMSOFT '19, New York, NY, USA, 13–18 Oct. 2019. ISBN 978-1-4503-6924-4. doi: 10.1145/3349568.3351551

- A. Grunwald, M. Schaarschmidt, and C. Westerkamp. LoRaWAN in a rural context: Use cases and opportunities for agricultural businesses. In P. Roer, editor, *Proceedings of the Mobile Communication-Technologies and Applications*; 24. ITG-Symposium, ITG-Fachbericht, pages 134–139. VDE-Verlag GmbH, Osnabrück, Germany, 15–16 May 2019

Book Chapters

- M. Schaarschmidt, M. Uelschen, and E. Pulvermüller. Towards power consumption optimization for embedded systems from a model-driven software development perspective. In H.-G. Fill, M. van Sinderen, and L. A. Maciaszek, editors, *International Conference on Software Technologies*, volume 1622 of Communications in Computer and Information Science, pages 117–142, Cham, Switzerland, 18 July 2022. Springer International Publishing. ISBN 978-3-031-11513-4. doi: 10.1007/978-3-031-11513-4_6

- M. Schaarschmidt, M. Uelschen, E. Pulvermüller, and C. Westerkamp. Energy-aware pattern framework: The energy-efficiency challenge for embedded systems from a software design perspective. In R. Ali, H. Kaindl, and L. A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, volume 1375 of Communications in Computer and Information Science, pages 182–207, Cham, Switzerland, 27 Feb. 2021. Springer International Publishing. ISBN 978-3-030-70006-5. doi: 10.1007/978-3-030-70006-5_8

Others

- M. Schaarschmidt, C. Fuhrmann, M. Uelschen, C. Westerkamp, and E. Pulvermüller. Energieeffiziente Entwurfsmuster für das Internet der Dinge - Möglichkeiten und Perspektiven für Single- und Multicore. In *Tagungsband Embedded Software Engineering Kongress*, pages 511–522, Sindelfingen, Germany, 3–7 Dec. 2018. (in German)

Supervised Student Theses

- J. Budde. Entwurf und Entwicklung einer Hardware-in-the-Loop Plattform zum Rapid-Prototyping modellbasierter Software. Master's thesis, Faculty of Engineering and Computer Science, Osnabrück University of Applied Sciences, 2022. (in German)

- S. Balzer. Entwicklung eines Model-In-The-Loop-Ansatzes zur Energieoptimierung von Sensorknoten für das Internet of Things. Bachelor's thesis, Faculty of Engineering and Computer Science, Osnabrück University of Applied Sciences, 2020. (in German)

In addition, the author has contributed to the scientific community through his work as a research associate with the following publications:

Conferences (Peer-Reviewed)

- C. Westerkamp, A. Grunwald, and M. Schaarschmidt. LoRaWAN, NB IoT and other radio networks for agricultural applications. In P. Roer, editor, *ITG-Fb. 304: Mobilkommunikation – Technologien und Anwendungen*, ITG-Fachbericht, pages 127–130. VDE-Verlag, Osnabrück, Germany, 18–19 May 2022. ISBN 978-3-80075873-9

- L. Huning, T. Osterkamp, M. Schaarschmidt, and E. Pulvermüller. Seamless integration of hardware interfaces in UML-based MDSE tools. In *Proceedings of the 16th International Conference on Software Technologies*, volume 1 of ICSOFT '21, pages 233–244, Online Streaming, 6–8 July 2021. INSTICC, SciTePress. ISBN 978-989-758-523-4. doi: 10.5220/0010575802330244

- M. Haverkamp, A. Grunwald, C. Westerkamp, and M. Schaarschmidt. Weitverkehrsfunkvernetzung für landwirtschaftliche Anwendungsfälle: LoRaWAN und NB-IoT für Unterflursensoren im Precision Farming. In M. Gandorfer, A. Meyer-Aurich, H. Bernhardt, F. X. Maidl, G. Fröhlich, and H. Floto, editors, *40. GIL-Jahrestagung, Digitalisierung für Mensch, Umwelt und Tier*, pages 97–102, Weihenstephan, Freising, Germany, 17–18 Feb. 2020. Gesellschaft für Informatik e.V. (in German)

- T. Thurow, M. Schaarschmidt, and C. Westerkamp. Funkbasierte 3D-indoorlokalisierung unter der Verwendung des Chan-Ho-Algorithmus. In P. Roer, editor, *ITG-Fb. 278: Mobilkommunikation*, ITG-Fachbericht, pages 69–74. VDE-Verlag, Osnabrück, Germany, 16–17 May 2018. ISBN 978-3-80074577-7. (in German)

- D. Pieper, M. Schaarschmidt, and C. Westerkamp. Kontextbezogene Verbindungstypanalyse für webbasierte Videokonferenzen in HTML5. In P. Roer, editor, *ITG-Fb. 258: Mobilkommunikation*, ITG-Fachbericht, pages 107–112. VDE-Verlag GmbH, Osnabrück, Germany, 7–8 May 2015. ISBN 978-3-80073937-0. (in German)

Book Chapters

- M. Schaarschmidt, C. Westerkamp, and H. Knöchel. Anbindung von Software-Agenten an Sensorknoten und mobile Systeme. In B. Vogel-Heuser, editor, *Softwareagenten in der Industrie 4.0*, chapter 7, pages 125–148. De Gruyter Oldenbourg, Berlin, Germany, 2018. ISBN 978-3-11052458-1. doi: 10.1515/9783110527056-007. (in German)

Workshops

- M. Schaarschmidt, C. Westerkamp, A. Hennewig, D. Pieper, H. Speckmann, and W. Bisle. Content adaptive signal compression for remote SHM and NDT operation. In F.-K. Chang and F. Kopsaftopoulos, editors, *Proceedings of the 10th International Workshop on Structural Health Monitoring*, IWSHM '15, Stanford, CA, USA, 1–3 Sept. 2015. Destech Publications. doi: 10.12783/SHM2015/148

Posters

- D. Kuemper, E. Reetz, M. Schaarschmidt, M. Fischer, E. Pulvermueller, and R. Toenjes. Test framework for IoT-based services - a knowledge driven approach. In *2014 European Conference on Networks and Communications*, EuCNC '14, Bologna, Italy, 23–26 June 2014. ISBN 978-1-4799-5280-9

List of Acronyms

AADL	Architecture Analysis & Design Language
ADC	Analog-to-Digital Converter
ALF	Action Language for Foundational UML
API	Application Programming Interface
ASM	Assembly
ATL	Atlas Transformation Language
CAN	Controller Area Network
CIM	Computing Independent Model
CMOS	Complementary Metal-Oxide-Semiconductor
CMSIS	Common Microcontroller Software Interface Standard
CPU	Central Processing Unit
CSV	Comma-separated Values
DAC	Digital-to-Analog Converter
DCP	Distributed Co-Simulation Protocol
DDD	Domain-driven Design
DFS	Dynamic Frequency Scaling
DIP	Dependency Inversion Principle
DMA	Direct Memory Access
DMAD	Direct Memory Access Delegation
DSL	Domain-specific Language
DPA	Direct Power Analysis
DVFS	Dynamic Voltage Scaling
DVS	Dynamic Voltage and Frequency Scaling
EAS	Energy-aware Sampling
EBC	Event-based Computing
EBNF	Extended Backus–Naur Form
EMF	Eclipse Modeling Framework
FLPA	Functional-level Power Analysis

FPGA Field-programmable Gate Array
FSM Finite State Machine
fUML Foundational UML
GPO General Purpose Output
GPIO General Purpose Input/Output
GRM Generic Resource Model
GUI Graphical User Interface
HAL Hardware Abstraction Layer
HiL Hardware-in-the-Loop
HRM Hardware Resource Modeling
I^2C Inter-Integrated Circuit
id Identifier
IIoT Industrial Internet of Things
ILPA Instruction-level Power Analysis
ISR Interrupt Service Routine
IoT Internet of Things
IPA Indirect Power Analysis
IP Intellectual property
JSON JavaScript Object Notation
LED Light-emitting Diode
LoRa Long Range
LoRaWAN Long Range Wide Area Network
LPWAN Low Power Wide Area Network
MARTE Modeling and Analysis of Real-Time and Embedded systems
MBD Model-based Development
MBE Model-based Engineering
MBT Model-based Testing
MCU Microcontroller Unit
MDA Model-driven Architecture
MDD Model-driven Development
MDE Model-driven Engineering
MDSD Model-driven Software Development
MiL Model-in-the-Loop
MOF Meta Object Facility
MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor
MPU Microprocessing Unit

NB-IoT Narrowband Internet of Things
NFC Near Field Communication
NFR Non-functional Requirement
NFP Non-functional Property
OCL Object Constraint Language
OMG Object Management Group
PAP Power Analysis Profile
PiL Processor-in-the-Loop
PIM Platform Independent Model
POSIX Portable Operating System Interface
PSM Platform Specific Model
PWM Pulse-width Modulation
QA Quality Assurance
QoS Quality of Service
QoS&FT UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms
RAM Random-access Memory
ReqIF Requirements Interchange Format
ROM Read-only Memory
RPC Remote Procedure Call
RQ Research Question
RTC Real-time Clock
RTL Register Transfer Level
RX Receive
SDXP Simulation Data eXchange Protocol
SI International System of Units
SiL Software-in-the-Loop
SMiL Software-Model-in-the-Loop
SoC System-on-Chip
SPI Serial Peripheral Interface
SPTP UML Profile for Schedulability, Performance, and Time Specification
SUT System Under Test
SysML Systems Modeling Language
TCP Transmission Control Protocol
TTN The Things Network
TVL Tag Value Language

TX	Transmit
UC²E	Unit for Central Control and Estimation
UART	Universal Asynchronous Receiver Transmitter
UML	Unified Modeling Language
UTP2	UML Testing Profile 2
UUID	Universally Unique Identifier
VSL	Value Specification Language
XiL	X-in-the-Loop
XML	Extensible Markup Language
XMI	XML Metadata Interchange

List of Figures

List of Listings

List of Tables

List of Symbols

Electrical Fundamentals

Notation	Description	Page
α	Activity factor	19
C_L	Load capacitance of the CMOS logic	19
E	Electric energy	19
f	Operating or clock frequency	19
I	Electric current	18
I_{leak}	Leakage current	19
I_{short}	Short-circuit current	18
P	Electric power	18
P_{dyn}	Dynamic power consumption (in CMOS cicuits)	18
P_{short}	Short-circuit power consumption (in CMOS circuits)	18
P_{static}	Static power consumption (in CMOS circuits)	18
P_{switch}	Switching power consumption (in CMOS circuits)	18
Q	Electric charge	17
t	Time	18
U	Voltage	18
V_{dd}	Supply voltage	18

Model Transformation

Notation	Description	Page
fm	Formalism	40
m	Model	40
mm	Metamodel	40
R	Mapping rules	40
s	System	40

Scenarios and Energy Bugs

Notation	Description	Page
C_i	Hardware component i	85
E_{C_i}	Energy consumption of hardware component i	85
E_{qu}	Energy available for a period T	85
E_S	Energy consumption of a SUT	85
I_{dmax}	Maximum current demand for a point in time	85
P_{C_i}	Electric power consumption of hardware component i	85
P_{tmS_i}	Scenario execution probability	84
P_S	Electric power consumption of a SUT	85
$S_{measure}$	Scenario for the measurment phase	87
S_{sleep}	Scenario for the sleep phase	87
$S_{transmit}$	Scenario for the transmit phase	87
S	Scenario	81
S_{tc}	Scenario of a test case	82
$\overline{S}_{tc}$	Scenario of a test case with periodic execution	83
$\widehat{S}_{tc}$	Scenario of a test case containing probabilities	83
$t_{measure}$	Time to end execution of $S_{measure}$	87
t_{sleep}	Time to end execution of S_{sleep}	87
$t_{transmit}$	Time to end execution of $S_{transmit}$	87
T	Time period	83
tc	Test case	82

Software Design Pattern Framework

Notation	Description	Page
c	Duration (time)	97
CP	Computing power	97
D	Power cycle	97
D'	Relaxed power cycle	97
E_{add}	Addtional energy consumption of design patterns	110
E_{dma}	Energy consumption of the DMAD design pattern	106
$E_{interrupt}$	Energy consumption for interrupts (EAS design pattern)	100
$E_{monitor}$	Energy consumption of the PowerMonitor design pattern	103
E_{normal}	Energy consumption without design pattern	98
$E_{polling}$	Energy consumption for polling (EAS design pattern)	100
$E_{relaxed}$	Reduced energy consumption (EAS design pattern)	98
E_{save}	Energy savings of design patterns	110
EB_P	Energy balance	94
η_C	Current demand cut-off factor	94
η_P	Effort-saving ratio	94
f_c	Control flow	102
f_{max}	Maximum frequency of the signal	96
f_s	Sequentially performed proportion of an algorithm	113
f_{sample}	Sampling frequency	96
p	Number of CPU cores	113
P_i	Power state i	97
S	Speedup	113
t_{hit}	Duration of a hit (EBC design pattern)	100
$t_{interrupt}$	Duration of an interrupt (EBC design pattern)	100
t_{miss}	Duration of a miss (EBC design pattern)	100
T_P	Parallel execution time	113

Notation	Description	Page
T_S	Sequential execution time	113
T	Time period	97
T'	Relaxed time period	97

Hardware Modeling

Notation	Description	Page
A_{hc}	Attributes of a hardware component model	121
δ	Initial power state	119
E_{es}	Energy consumption of an embedded system	120
EM_{hc}	Energy model of a hardware component model	121
I_δ	Electric current consumption of a transition	119
I_s	Electric current consumption of a state	119
O_{hc}	Operations of a hardware component model	121
P_{es}	Power consumption of an embedded system	120
s_0	Initial power state	119
Σ	Input alphabet	119
S_p	Finite set of power states	119
T_s	Execution time of a state	119
T_t	Execution time of a transition	119
V_c	Supply voltage of a hardware component	119

Evaluation

Notation	Description	Page
Δt	Time difference	194
Q_1	Lower quartile	193
Q_3	Upper quartile	193
S_a	Scenario for the active phase of the system	185
S_s	Scenario for the sleep phase of the system	185
S_t	Scenario for the transmit phase of the system	185
S_{tl}	Test lab scenario	185
σ	Standard deviation	193
t_D	Total time delay	192
t_{DC}	Time delay for communication	
t_{DI}	Time delay for interpretation	
t_{DP}	Time delay for message processing	193
$\tilde{x}$	Median	193

Appendix

A Supplemental Background

This chapter provides additional information as advanced knowledge for background discussed in Chapter 2 (p. 17 ff.). Supplemental background on architectural layers and metamodels of MDA as the specific interpretation of MDD relying on OMG standards is presented in Section A.1, while Section A.2 presents a comparison of MDD tools. Section A.3 gives a brief introduction of UML and UML diagrams used in this thesis.

A.1 Model-driven Architecture (MDA)

This section gives a brief overview of MDA as an OMG-specific implementation of MDD introduced in Section 2.5 (p. 37 ff.) with the specified architectural layers in introduced Section A.1.1 and metamodels discussed in Section A.1.2.

A.1.1 Architectural Layers

A key concept of MDA is the three-layer architecture with different levels of abstraction to achieve a separation of concerns. For each layer, MDA introduces a specific type of model.

The *Computing Independent Model (CIM)*, as the most abstract layer, specifies business activities and requirements of the system without referring to the implementation. It may also contain parts that are not mapped to software implementations. Due to the focus on business processes, the CIM is also denoted as a business or domain model [56, 272].

The *Platform Independent Model (PIM)* is part of the second layer of abstraction in which logical aspects of the system, such as the structure and behavior, are addressed. Compared to CIM, PIM describes only parts of the CIM based on formal modeling languages and contains the formal specification of the architecture, structure, and functionality of a system regarding defined requirements while being independent of any specific implementation technologies. The provided level of abstraction enables the PIM to be mapped to one or more concrete implementation platforms without the need for adjustments [56, 272, 369].

As part of the lowest layer, the *Platform Specific Model (PSM)* contains all information regarding the structure and behavior of the systems. It utilizes specific technologies to implement the functionalities defined by a PIM for a specific platform. On this level, MDD tools can use PSMs for simulations or code generation approaches to transform a PSM into generate platform-specific source code. Figure A.1 shows the relation between CIM, PIM, and PSM.

Besides the aforementioned layer of abstraction and associated model representations, MDA envisions a complete development process, from the description of business processes to executable and platform-specific software applications. For such a process, model transforma-

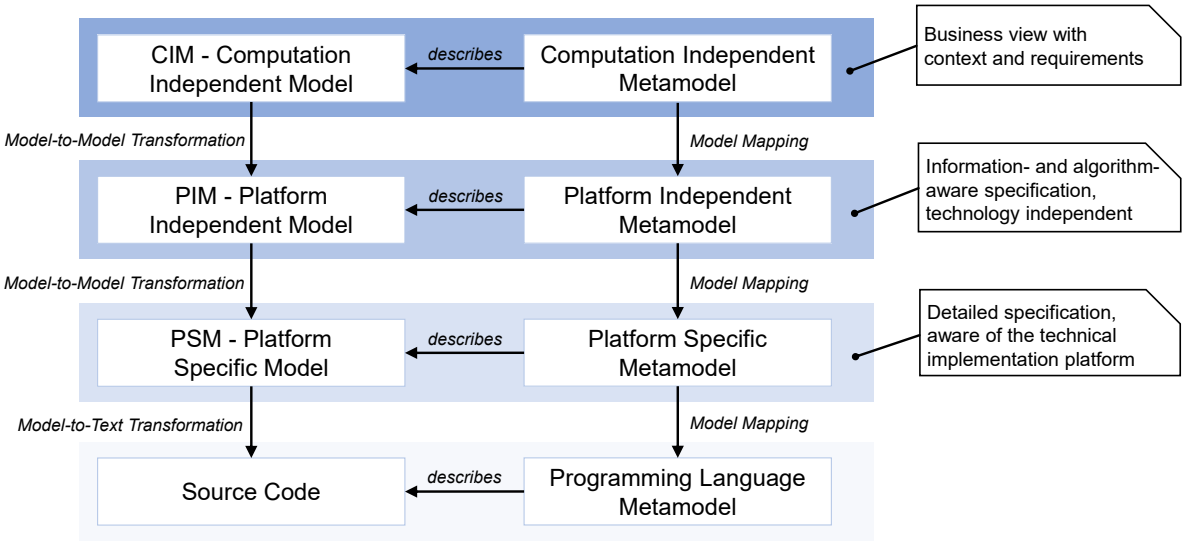


Figure A.1: Model abstractions defined in MDA, adapted from [56].

tions are needed. However, to achieve an automatic transformation between different models, detailed and formal model descriptions of the source and target models are required. Such model descriptions are denoted as metamodels.

A.1.2 Metamodels

To perform a model transformation as described in Section 2.5.2 (p. 40 ff.), modeling languages have to be specified in a formal manner. With models as primary artifacts, MDD and MDA aim to follow the *"everything is a model"* principle as an analogy to the *"everything is an object"* of object-oriented software development and *"everything is a file"* as one of the defining features of the Unix operating system [48, 56]. As a logical consequence, so-called metamodels are used to define a modeling language [56]. Metamodels, however, may, in turn, be described by meta-metamodels. Due to their level of abstraction, meta-metamodels are often able to define themselves with the same language elements they provide. As shown in Figure A.2, the concepts and relations of models, metamodels, and meta-metamodels can be expressed by using the four-layered architecture of the MOF standard, introduced by the OMG [56, 272]:

- Level M0 defines a real object or instance of a model located at level M1.
- Level M1 defines the structure and behavior of objects located at M0.
- The metamodel describing the concepts for models at level M1 is located at level M2.
- The meta-metamodel that specifies the concepts used at level M2 is located at level M3.

The MOF is typically used at M3 to define a metamodel such as UML. Besides the MOF, Ecore, as part of the *Eclipse Modeling Framework (EMF)*, may also be used at level M3, which is tailored to Java for implementation purposes [56]. A transformation may also be a model itself and, therefore, an instance of a transformation metamodel. For example, the ATL may be used to specify rules for the transformation between two models, whereas the ATL metamodel defines the syntax of the ATL language [56].

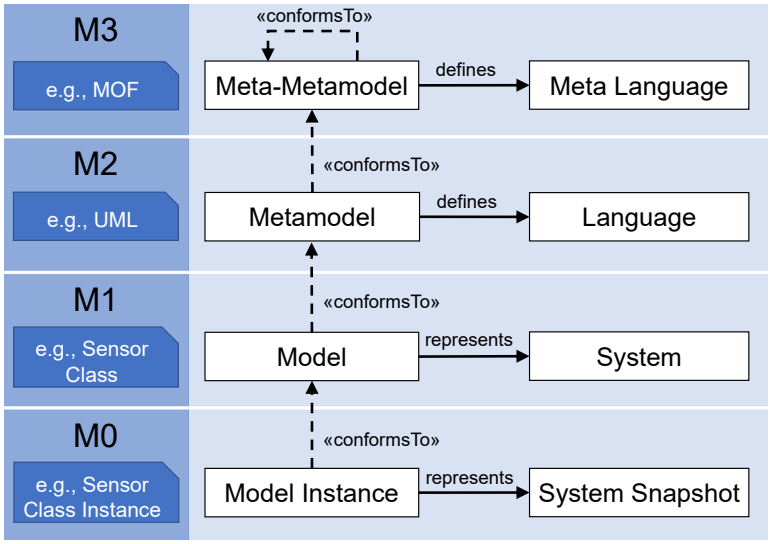


Figure A.2: Four-layered MOF architecture showing the relation between objects, models, metamodels, and meta-metamodels, adapted from [56].

A.2 Tool Support in Model-driven Development (MDD)

An important aspect of MDD is the availability of tools to support the MDD process in terms of modeling, transformation of models, and source-code generation. Over the previous decades, various types of MDD tools have been developed, including open-source and proprietary solutions. However, it is important to distinguish between drawing and modeling tools. According to Brambilla et al. (2017) [56], an MDD tool must provide at least some of the following functionalities:

- An MDD tool guarantees a minimum level of semantic meaning and model quality to ensure compliance with metamodels.
- An MDD tool offers appropriate functionalities for model transformations, e.g., model-to-model, model-to-text, and code generation, cf. Section 2.5.2 (p. 40 ff.).
- An MDD tool uses APIs to export and manipulate models and to customize code generation.
- An MDD tool provides semantic-aware export formats such as the XMI [273].

For instance, modeling tools may use a concrete textual syntax to specify models without drawing support. With drawing tools, developers may be able to create UML-like diagrams, but these tools often lack a semantic description of the drawn models. As mentioned in [7], Microsoft Visio [251] is one of the most widely used modeling tools. However, since Microsoft Visio does not include any of the aforementioned functionalities, it is not considered as an MDD tool in this thesis. Well-known MDD tools that provide some of the abovementioned functionalities are Rhapsody [164], MagicDraw [90], Visual Paradigm [408], Enterprise Architect [364], and MathWorks MATLAB [380] as examples of commercial MDD tools, and Papyrus [104] as an example of an open-source solution. With the exception of MathWorks

MATLAB, the aforementioned MDD tools provide software developers with multiple editors to define UML models graphically. Table A.1 compares well-known MDD tools and their

MDD Tool (Version)	Supported Languages			
	UML	*SysML*	*MARTE*	*fUML(ALF)*
Rhapsody (9.0.1)	✓	✓	✓	✗
MagicDraw (19.0 LTR SP4)	✓	✓	✓	✓
Visual Paradigm (17.0)	✓	✓	✗	✗
Enterprise Architect (16.0)	✓	✓	✓	✗
Papyrus (6.2.0)	✓	✓	✓	✓

Table A.1: Comparison of well-known MDD tools for UML-based modeling languages. The symbol ✓ indicates that the language is supported by the MDD tool, while the symbol ✗ indicates no language support.

support of UML-based modeling languages introduced in Section 2.5.1 (p. 38 ff.). Besides basic UML, languages such as SysML and MARTE are accepted and supported by well-known MDD tools. On the other hand, the support of fUML [281] and the corresponding ALF [277] can be considered insufficient, as shown in Table A.1.

Table A.2 briefly summarizes the features provided by MDD tools for UML. All introduced MDD tools provide an environment for simulating selected UML diagram types, such as state machine and activity diagrams. The most important languages for embedded systems, C and C++, are well supported by the MDD tools in Table A.2, although the level of generation varies. Furthermore, all MDD tools provide a model exchange based on XMI and API support. For example, Rhapsody, MagicDraw, and Enterprise Architect offer a Java-based API that allows software developers to interact with UML models and the code generation process. Further comparisons between MDD tools can be found in [288, 330].

MDD Tool (Version)	Simulation	Code Generation	Model Exchange	API Support
Rhapsody (9.0.1)	AD, SMD	C, C++, Java, Ada	XMI	✓
MagicDraw (19.0 LTR SP4)	AD, SMD	Java, C#, C++	XMI	✓
Visual Paradigm (17.0)	AD, SD	C++, C#, Java, VB, PHP, Ada, ActionScript	XMI, XML	✓
Enterprise Architect (16.0)	AD, ID, SMD	C, C++, Java, Ada	XMI	✓
Papyrus (6.2.0)	AD, CD, CSD, SMD	C++, Java	XMI	✓

AD = Activity Diagram, CD = Class Diagram, CSD = Composite Structure Diagram, ID = Interaction Diagram, SD = Sequence Diagram, SMD = State Machine Diagram

Table A.2: Feature comparison of MDD tools.

A.3 Unified Modeling Language (UML)

This section briefly introduces UML as the underlying specification MARTE is based on. It also provides supplemental background on the notation of the diagrams used to define hardware and software models in Chapters 5 to 7. UML specifies a set of fourteen different diagrams for a graphical specification of the software application from different perspectives. Each diagram

type can be interpreted as a different view for specific parts of the software application model. Figure A.3 gives an overview of the diagram types defined in the UML specification [275], which

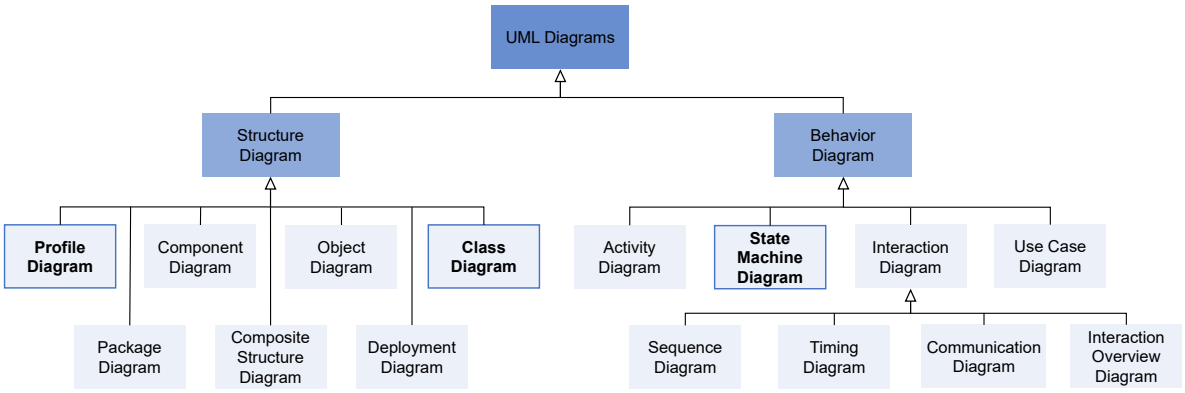


Figure A.3: Overview of UML diagram types, adapted from [275]. Diagrams used in this thesis are framed blue and with bold characters.

can be categorized into structural and behavioral diagrams. Structural diagrams describe the overall structure and the relationship between different parts of the software application, e.g., hierarchy and inheritance of classes. Behavioral diagrams specify the flow-related behavior of objects, e.g., instances of classes changing over time.

Besides the graphical notation in the form of diagrams as a human-readable notation, the UML specification also provides a metamodel (cf. Section A.1.2, p. 264 f.) to specify the abstract syntax of the UML and to add semantic meaning to the diagram descriptions. The abstract syntax defines modeling concepts, their relationships, and a set of rules for combining these concepts to create UML models. Additionally, the UML specification defines semantics that describes how UML concepts can be realized and interpreted by MDD tools. This enables features such as model transformation (cf. Section 2.5.2, p. 40 ff.), verification, simulation, code generation, and modification of UML models via programming languages using the provided API of MDD tools (cf. Section A.2, p. 265 ff.).

The blue-framed diagram types in Figure A.3 are used for different concepts in this thesis and are introduced in the following sections with a focus on software development. This section does not intend to cover the full specification of each diagram type. A detailed description may be found in the specification [275] and literature such as [125, 347]. Section A.3.1 introduces the UML class diagram. Section A.3.2 describes the UML profile diagram as another structural diagram type mainly used to define extensions of the UML metamodel. In Section A.3.3, UML state machine diagrams are introduced.

A.3.1 Class Diagram

A UML class diagram describes the static structure of a software application. Among all diagrams presented in Figure A.3, the UML class diagram is the best-known and most-used diagram type of the UML specification applied in different phases of the software development process with varying levels of detail and abstraction [125, 347]. The main elements of a UML class diagram are classes, which are modeled with their properties and operations. These structural characteristics are also denoted as features of a class.

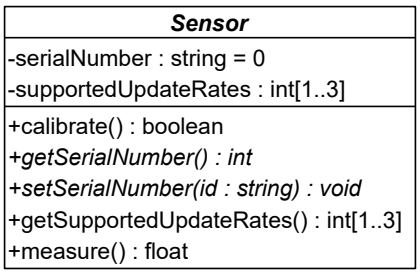


Figure A.4: An example of a single class in the UML 2.5.1 notation [275].

Figure A.4 shows the graphical representation of an example UML class as a box with three compartments that define, from top to bottom, the name, attributes, and operations of the class. The syntax of attributes in the middle compartment of Figure A.4 consists of a visibility marker, the name of the attribute, the data type, and optionally a multiplicity and a default value. The visibility defines the access permission of the specific attribute. In UML, the main permissions are private, public, and protected. Attributes marked as private (`-`) are only accessible within the object itself. Access by objects of the same class and sub-classes is denoted as protected (`#`), and the marker `+` indicates that any object can access the attribute directly. The data type of an attribute can be a primitive data type (e.g., integer) or a composite data type (e.g., another class). Additionally, rectangular brackets after the data type define the multiplicity of an attribute if it, e.g., represents a set or list. For instance, with the notation `[minimum..maximum]`, an attribute may contain at least one and a maximum of three elements `[1..3]`, exactly three elements `[3]`, or an indefinite amount of elements `[*]` that may change during execution. Equal signs are used to define optional default values.

Operations are characterized by a visibility marker, a name, parameters, and the data type of the return value. In a class diagram, the name of the operation is followed by a list of parameters in parentheses which may be empty if no parameters are required. Parameters in the parameter list are represented similarly to attributes, e.g., by defining the name and data type. The return type is also optional and specifies the data type of the value returned as a result of an operation call.

Besides the elements of the software application, a UML class diagram also describes the relationship between those elements. Since the relationships between classes do not change over time, this diagram type is considered static. Figure A.5 shows an example of a UML class diagram and introduces additional semantics of the UML specification for class diagrams. Relationships between classes are defined as lines. In UML, relationships are distinguished graphically by the design of the line and the use of symbols, e.g., arrows, at the end of the lines. Relationships covered in this thesis are associations, shared aggregations, compositions, inheritances, and realizations [125, 347]:

Associations describe the binary relationship between two classes in which one class instance can start actions on another. Graphically, associations are visualized by a solid line between those two classes. With the use of directed edges, associations can be further specified to indicate navigability between two instances of a class which can be unidirectional and bidirectional. The unidirectional navigability indicates that an instance of a class has knowledge about the instance of the other class and can access its public attributes and operations. This is expressed with an open arrow at the end of the class whose attributes

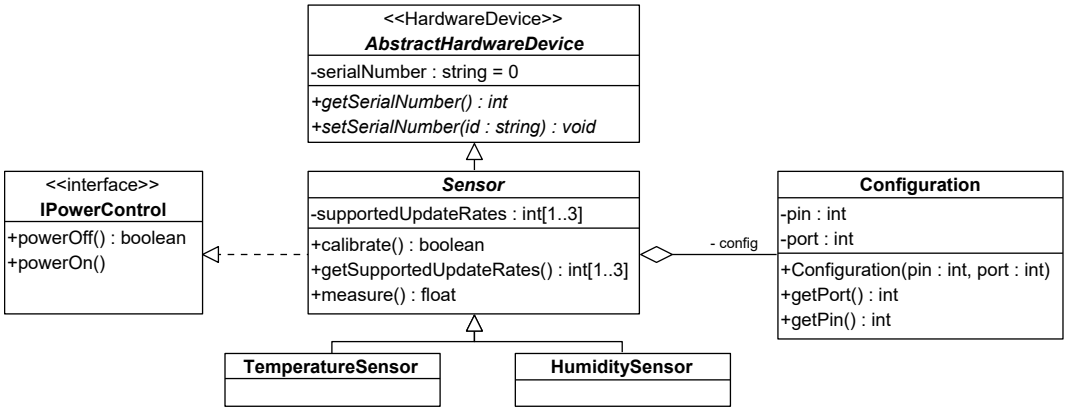


Figure A.5: An example of a UML class diagram in the UML 2.5.1 notation [275].

and operations can be accessed. For bidirectional navigability, open arrows are placed at both ends of the line.

Shared aggregations and ***compositions*** are special interpretations of associations and define *part of* relationships between two classes.

- A shared aggregation expresses a weaker form of the *part of* relationship where parts belong to the *whole* but can also exist independently of the *whole*. The notation of a shared aggregation in a UML class diagram is defined as a line with a diamond on the end of the class representing a *whole*. For example, the relationship between the classes `Sensor` and `Configuration` in Figure A.5 is defined as a shared aggregation where the instance of a `Configuration` is part of a `Sensor` instance but can also exist without a specific `Sensor` instance, e.g., if the same configuration is (re-)used for other sensors.

- Compositions define a stronger bond between the *whole* and the *part*, graphically expressed as a filled diamond. In a composition, the *part* can not exist independently without the *whole*.

Associations, shared aggregation, and compositions can be extended by adding multiplicities on one or both ends. In Figure A.5, the multiplicity between `Sensor` and `Configuration` is set to one at both ends because a `Sensor` instance only has one `Configuration`, and a `Configuration` instance is part of one specific `Sensor` instance.

An ***inheritance*** describes a directed relationship between classes in which a specific class inherits the attributes and operations from another class, e.g., the superclass. An inheritance is visualized as a solid line with a triangle arrowhead pointing to the superclass. Figure A.5 shows two inheritances between `AbstractHardwareDevice` and `Sensor` and between `Sensor`, `TemperatureSensor`, and `HumiditySensor`.

A ***realization*** defines a type of dependency where a class implements the behavior of another class. Graphically, a realization is represented with a dashed line and a triangular arrowhead. An interface realization is shown in Figure A.5 between the classes `IPowerControl` and `Sensor`. The instance of `TemperatureSensor` and `HumiditySensor` implement the operations defined by `IPowerControl`. Furthermore, *stereotypes*, represented with angular brackets, can be added to classes refining their meaning. For example, the name compartment of class `IPowerControl` in Figure A.5 has been extended with the stereotype «`interface`».

This changes the semantic meaning of the defined class and turns the rectangle in its meaning from a class to an interface. In object-oriented programming languages like C++, programs are based on class constructs, and classes become objects at runtime. Classes extended with the stereotype «`interface`» are interpreted as communication interfaces to be implemented by other classes. Therefore, no instances of these classes exist at runtime. Stereotypes are discussed in more detail in the following Section A.3.2.

A.3.2 Profile Diagram

A UML profile diagram specifies stereotypes as part of a profile. As an extension mechanism, stereotypes are used to create new model elements by extending the vocabulary of UML. In contrast to UML class diagrams where stereotypes are applied to classes, as shown in Figure A.5 (p. 269), the UML profile diagram describes the structure and content of stereotypes and the type of UML metamodel element to which they can be applied. Due to this, UML profile diagrams are located at level M2 (cf. Section A.1.2, p. 264 ff.). An exemplary profile diagram is illustrated in Figure A.6. The visual representation of a UML profile diagram is similar to the UML class diagram. Stereotypes are described like classes with the additional keyword `stereotype` above the name in the first compartment. In the second compartment, properties and their data types are defined.

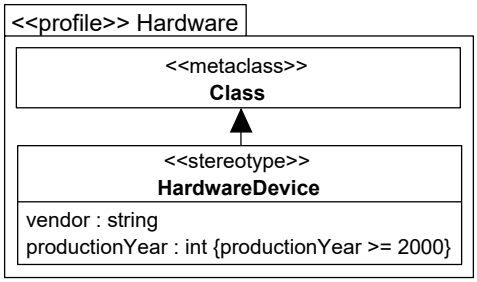


Figure A.6: Example of a UML profile diagram in the UML 2.5.1 notation [275].

Additionally, constraints for properties may be described. If a stereotype is applied to a UML element, the properties are denoted as key-values pars, also referred to as *tagged values*. The second element of UML profile diagrams are metaclasses, which are represented as a box with a name and the additional keyword `metaclass`. Solid lines with a filled arrow indicate which metaclass is extended by a specific stereotype. In general, a stereotype may extend one or more metaclasses. Figure A.6 shows an exemplary UML profile diagram with a stereotype `HardwareDevice` consisting of two attributes, `vendor` as a string and `productionYear` as an integer. The newly defined stereotype can be applied to UML class meta elements. Note that the attribute `productionYear` has a constraint where values only higher or equal to 2000 are accepted. In Figure A.5, the classes `TemperatureSensor` and `HumiditySensor` have been extended with the `HardwareDevice` stereotype via inheritance to add information about the vendor and year of production as additional metadata.

A.3.3 State Machine Diagram

Discrete event-driven behavior can be modeled by using a finite state-machine formalism. The form of finite state automata used in the UML [275] is based on the object-oriented

statecharts formalism presented in [155]. The UML specification [275] distinguishes between two state machine types: *behavioral state machines* and *protocol state machines*. *Behavioral state machines* specify the discrete behavior of classifiers, such as class instances, by modeling system states and state transitions between states. State transitions are executed as a reaction to events that may occur at certain points in time. On the other hand, *protocol state machines* are used to express valid event sequences and the order of, e.g., operations calls, to which a system must conform. Additionally, states of a *protocol state machine* do not have any activities within the states. Instead, activities are only executed during transitions. In this thesis, however, only diagrams for *behavioral state machines* are discussed. In the following, characteristics and types of states and transitions are described and illustrated with an example.

States

The UML specification distinguishes between two types of states, namely *pseudo states* and *real states*. *Pseudo states* do not have any internal activities and are transient, which means that the state machine cannot remain in such states. The initial state is an example of a *pseudo state* graphically represented as a filled black circle with no incoming edges. Normal states and the final state are considered as *real states*. A normal state is represented in a UML state machine diagram as a rectangle with rounded corners divided into two compartments. The upper compartment contains the name of the state. In the lower compartment, internal activities in the form of `keyword/activity` are specified, which may consist of multiple actions. The keywords `entry` and `exit` suggest that the activity must be executed when a state is entered or left. Activities after the keyword `do` are executed as long as the current state is active. Final states are represented by a circle containing a smaller filled circle and mark the end of a state sequence.

State Transitions

The change from a source state S_{source} to a target state S_{target} is denoted as state transition, i.e., $S_{source} \rightarrow S_{target}$. Graphically, state transitions are represented by a solid line and an arrowhead that points from the source state S_{source} to the target state S_{target}. State transitions may have a set of properties graphically specified on top of the transition arrow as $E(P)[G]/A$, with E as an event, P as an optional parameter list of the event E, G as a guard, and A as the activity executed during the transitions. A guard represents a condition as a boolean expression evaluated as true or false if an event specified for a transition occurs at a point in time. If the guard is evaluated to true, the current state activities are terminated, the exit activity performed, and the transition to the target state S_{target} executed. If the guard is evaluated to false, no transition is performed. In the UML specification, several types of events are defined, including *signal event*, *call event*, *time event*, *change event*, *any receive event*, and *completion event* [347]. *Signal events* are asynchronous events for which the sender does not wait for a response. *Call events* represent operation calls where the name of the event is equal to the name of an operation. *Time events* enable transitions to be executed after a certain time window, while *change events* are consistently evaluating a condition. If the condition is satisfied, the change event is dispatched. If the transition should be executed when an event of any type occurs, the *any receive event* type may be applied. As the last type of event, the *completion event* is dispatched if the `entry` and `do` activities have been completed. State transitions can be distinguished between *internal transitions* and *external transitions*. While

external transitions are transitions between states, internal transitions trigger activities, but they do not invoke a state transition, and therefore no entry and exit activities are performed. A special form of external transition is called *self-transition*, where the source and target state are equal, i.e., $S_{source} == S_{target}$. Unlike the internal transition, entry and exit activities are invoked when self-transitions are executed. If no guard or event is modeled for a transition, it is executed after the entry and the do activity have been performed.

Example

Figure A.7 shows an exemplary UML state machine with the three states `InitSystem`, `Measure`, and `Compare`. A filled black circle denotes the initial state, while the final state is illustrated by a circle with an additional inner-filled circle. Transitions in Figure A.7 have been extended with guards, e.g., `measuredVal>=100`, and actions, e.g., `setAlarm()`. For more information about the UML state machine diagram and UML diagrams in general, see [100, 125, 138, 347].

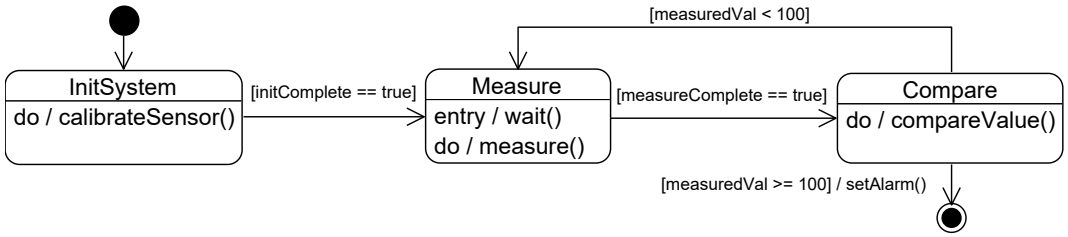


Figure A.7: Example of a UML state machine diagram in the UML 2.5.1 notation [275].

B Complete List of Tags for PAP Stereotypes

This section contains a complete list of tags for the stereotypes of the PAP profile developed in this thesis. Table B.1 summarizes existing and newly defined tags of the `HwAbstraction` stereotype, introduced in Section 5.3.3 (p. 127 ff.). Existing tags are introduced due to the inheritance of the MARTE stereotype `HwResource` but are not required for the power estimation process.

Tag	Description	Type
`description`	A textual description of the component.	NFP_String
`endPoints`	*HwResource* connection points.	HW_EndPoint[0..*]
`frequencies`[1]	Specifies a set of supported clock frequencies, cf. Section 5.2.2 (p. 119 f.).	NFP_Frequency[1..*]
`frequency`	Specifies the clock frequency.	NFP_Frequency[0..1]
`ownedHW`	Specifies the owned sub-*HwResources*.	HwResource[0..*]
`p_HW_Services`	Specifies the provided services.	HwResourceService[1..*]
`r_HW_Services`	Specifies the required services.	HwResourceService[0..*]
`supplyVoltage`[1]	Specifies the supply voltage.	NFP_Voltage[2]

Table B.1: Tags of the `HwAbstraction` stereotype (*HardwareAbstraction* package) with additional tags[1] and data types[2] introduced in this thesis that are not part of the MARTE specification [275].

Table B.2 summarizes the tags for the `HwBehavioralState` and `HwBehavioralTransition` stereotypes, introduced in Section 5.3.4 (p. 129 ff.). In addition to newly defined tags for the power analysis process, they contain tags due to the inheritance of the MARTE stereotypes `ResourceUsage` and `HwResourceService`.

Tag	Description	Data Type
`current`[1]	Electric current consumption of the state/transition.	NFP_ElectricCurrent[1..*][2]
`hasDynamicConsumption`[1]	True, if the electric current consumption is static, false otherwise.	NFP_Boolean
`hasDynamicExecutionTime`[1]	True, if the execution time is static, false otherwise.	NFP_Boolean
`HwResourceService` (HwPower Package)		
`consumption`	Consumed power.	NFP_Power
`dissipation`	Power dissipated, e.g., heat.	NFP_Power
`ResourceUsage` (GRM Package)		
`allocatedMemory`	Amount of memory that is demanded from or returned to the resource.	NFP_DataSize[*]
`execTime`	Execution time of the state/transition.	NFP_Duration[*]
`energy`	Amount of energy that will be permanently consumed.	NFP_Energy
`msgSize`	Amount of data transmitted by the resource.	NFP_DataSize[*]
`powerPeak`	The maximum power that is demanded and should be available from the resource.	NFP_Power[*]
`usedMemory`	Amount of memory that will be used from a resource.	NFP_DataSize[*]

Table B.2: Tags of the `HwBehavioralState` and `HwBehavioralTransition` stereotypes (*HardwareBehavior* package) with additional tags[1] and data types[2] introduced in this thesis that are not part of the MARTE specification [275].

C Model Transformation Example

This section provides additional information for the model transformation process as a part of the prototype implementation introduced in Section 6.1 (p. 140 ff.). Section C.1 provides a more coherent and complete sample output of the developed IBM Rhapsody plug-in (cf. Section 6.1.1, p. 140 ff.) for the dimmable LED example introduced in Section 5.3 (p. 124 ff.). Section C.2 provides supplemental mapping information between the model interchange format discussed in Section 6.1 (p. 140 ff.) and the representation of hardware component models in C++ as part of the UC^{2}E tool introduced in Section 6.4 (p. 155 ff.).

C.1 Model-to-Text Transformation

Listing C.1 shows the JSON-based description resulting from an exemplary model-to-text transformation of the dimmable LED hardware component model discussed in Section 5.3.3 to 5.3.5 (p. 127 ff.) with the model interchange format introduced in Section 6.1.1 (p. 140 ff.).

```
1  {
2     "HardwareComponentModels": [
3        {
4           "Name": "DimmableLED",
5           "Attributes": {
6              "brightnessLevel": {
7                 "id": "brightness",
8                 "dataType": "PercentageInteger"
9              }
10          },
11          "Settings": {
12             "supplyVoltage": "(3.3,V)"
13          },
14          "States": {
15             "OFF": {
16                "name": "Off",
17                "id": "11f299b3-6734-4ecb-8850-5352c84b3b0a",
18                "behavior":{
19                   "current": "(0,mA)",
20                   "execTime": "",
21                   "hasDynamicConsumption": false,
22                   "hasDynamicExecTime": false
23                }
24             },
25             "ON": {
26                "name": "On",
27                "id": "c5df30c6-6c6c-4a72-b6c9-345cc9834942",
28                "behavior":{
29                   "current": "((PAP.ATTR.brightness/100)*5,mA)",
30                   "execTime": "",
31                   "hasDynamicConsumption": false,
32                   "hasDynamicExecTime": false
33                }
34             }
35          },
36          "Transitions": {
37             "0": {
38                "name": "0",
39                "initialTransition": true,
```

```
                "fromState": "",
                "toState": "11f299b3-6734-4ecb-8850-5352c84b3b0a",
                "behavior":{
                   "current": "(0,mA)",
                   "execTime": "(0,ms)",
                   "hasDynamicConsumption": false,
                   "hasDynamicExecTime": false
                }
             },
             "1": {
                "name": "1",
                "initialTransition": false,
                "fromState": "11f299b3-6734-4ecb-8850-5352c84b3b0a",
                "toState": "c5df30c6-6c6c-4a72-b6c9-345cc9834942",
                "behavior":{
                   "current": "(0,mA)",
                   "execTime": "(0, ms)",
                   "hasDynamicConsumption": false,
                   "hasDynamicExecTime": false
                }
             },
             "2": {
                "name": "2",
                "initialTransition": false,
                "fromState": "c5df30c6-6c6c-4a72-b6c9-345cc9834942",
                "toState": "11f299b3-6734-4ecb-8850-5352c84b3b0a",
                "behavior":{
                   "current": "(0,mA)",
                   "execTime": "(0,ms)",
                   "hasDynamicConsumption": false,
                   "hasDynamicExecTime": false
                }
             }
          }
       }
    ]
}
```

Listing C.1: Exemplary model-to-text transformation from IBM Rhapsody to the JSON-based interchange format based on the dimmable LED hardware component model.

C.2 Model Mapping of Hardware Component Models

Table C.1 describes the element-wise mapping for the text-to-model transformation of hardware component models from the JSON-based interchange format (cf. Section 6.1.1, p. 140 ff.) to the C++-based hardware model of the UC^{2}E (cf. Section 6.4, p. 155 ff.). The model mapping between the interchange format and the UC^{2}E tool is essential for IPA and DPA. It is used for the evaluation in Chapter 7 (p. 175 ff.) to perform a power analysis and to derive energy traces.

JSON Element	C++ Element	Note
`['HardwareComponentModels'][i]`	–	`i` as the number of the i-th hardware component model in the JSON structure.
`i["Name"]`	`HwModel.baseName`	Name of the hardware component. `HwModel.name` contains the name of the instance and is set dynamically during simulation.
`i["Attributes"]["AttributeName"]`	`HwAttribute`	The data type is ignored since values in expressions are always numeric.
`i["Attributes"]["AttributeName"].id`	`HwAttribute.id`	
`i["Attributes"]["AttributeName"].value`	`HwAttribute.value`	
`i["Settings"]["id"]`	`HwSetting`	With `id = HwModel.id` and `(value,unit) = HwModel.value` and `HwModel.unit`.
`i['States']['stateId']`	`HWState`	
`i['States']['stateId'].name`	`HWState.name`	
`i['States']['stateId'].id`	`HWState.id`	
`i['States']['stateId']['behavior'].current`	`HWState.current`	
`i['States']['stateId']['behavior'].execTime`	`HWState.execTime`	
`i['States']['stateId']['behavior'] .hasDynamicConsumption`	`HWState .hasDynamicConsumption`	
`i['States']['stateId']['behavior'] .hasDynamicExecutionTime`	`HWState. hasDynamicExecutionTime`	
`i['Transitions']['transitionId']`	`HWTransition`	
`i['Transitions']['transitionId'].name`	`HWTransition.name`	
`i['Transitions']['transitionId'].initialTransition`	`HWTransition.isInitialTransition`	

Continued on next page

Table C.1 – continued from previous page

JSON Element	C++ Element	Note
`i['Transitions']['transitionId'].fromState`	`HWTransition.fromState`	Reference based on `HWState.id`.
`i['Transitions']['transitionId'].toState`	`HWTransition.toState`	Reference based on `HWState.id`.
`i['Transitions']['transitionId']['behavior'] .current`	`HWTransition.current`	
`i['Transitions']['transitionId']['behavior'] .execTime`	`HWTransition.execTime`	
`i['Transitions']['transitionId']['behavior'] .hasDynamicConsumption`	`HWTransition .hasDynamicConsumption`	
`i['Transitions']['transitionId']['behavior'] .hasDynamicExecutionTime`	`HWTransition .hasDynamicExecutionTime`	

Table C.1: Mapping of hardware component models by the UC^{2}E tool.

D Supplementary Information about Model-RPC

This section contains supplementary information about the specified *Model-RPC* communication protocol (cf. Section 6.5.4, p. 168 ff.). Section D.1 provides a definition and JSON schema description of the `configType` data structure introduced in Section 6.5.4 (p. 168 ff). It is used to configure the developed *Model-Testbeds*. In Section D.2, an exemplary OpenRPC schema specification for the UART `write` method is shown for the automatic validation of *Model-RPC* messages.

D.1 The `configType` Object Structure

In Table D.1, each property of the `configType` object is listed along with the corresponding data type and a brief description. The *values* field may contain a description of the value or provides a set of values that can be used for a specific data type.

Property		Type	Description	Values
name		String	Name of configuration	-
date		String	Creation date of configuration	-
gpio		Array	GPIOs objects to configure	Object with properties `id` and `mode`
	id	String	Id of GPIO	-
	mode	Number	Selected mode for the GPIO	0: Input, 1: Output, 2: ADC, 3: Interrupt (falling edge), 4: Interrupt (rising edge), 5: Interrupt on change
pwm		Array	PWM channel to configure	Objects with properties `id` and `state`
	id	String	Id of PWM	-
	state	Number	Enable/disable PWM channel	0: Off, 1: On
i2c		Array	I^2C interfaces to configure	Objects with properties `id` and `rate`
	id	String	Id of I^2C	-
	rate	Number	Transfer rate	0: 100 kBit/s, 1: 400 kBit/s, 2: 1,000 kBit/s
uart		Array	UART interfaces to configure	Objects with properties `id` and `baud`
	id	String	Id of UART	-
	baud	Number	Transfer rate	0: 9,600 Baud/s, 1: 14,400 Baud/s, 2: 19,200 Baud/s, 3: 38,400 Baud/s, 4: 57,600 Baud/s, 5: 115,200 Baud/s, 6: 128,000 Baud/s, 7: 256,000 Baud/s
spi		Array	SPI interfaces to configure	Objects with properties `id` and `rate`
	id	String	Id of SPI	-
	rate	Number	Configure word-size	4–16-Bit
can		Array	CAN interfaces to configure	Objects with properties `id`, `rate`, and `filter`
	id	String	Id of CAN interface	-
	rate	Number	Data rate	0: 250 kBaud/s, 1: 500 kBaud/s, 2: 1,000 kBaud/s
	filter	Array	List of filter IDs	

Table D.1: Properties of the *Model-RPC* `configType` data structure.

The following Listing D.1 provides a JSON schema description that can be used to validate `configType` objects.

```
{
   "$schema": "https://json-schema.org/draft/2020-12/schema",
   "id": "configType",
   "description": "ConfigType Definition.",
   "properties": {
      "name": {
         "type": "string",
         "description": "Name of the configuration.",
         "title": "Name"
      },
      "date": {
         "type": "string",
         "description": "Date of the configuration.",
         "title": "Date"
      },
      "gpio": {
         "type": "array",
         "description": "GPIO objects to configure.",
         "items": {
            "type": "object",
            "properties": {
               "id": {
                  "type": "string",
                  "description": "ID of the GPIO.",
                  "title": "id"
               },
               "mode": {
                  "type": "integer",
                  "description": "Selected mode of the GPIO.",
                  "title": "mode",
                  "minimum": 0,
                  "maximum": 4
               }
            },
            "required": [
            "id",
            "mode"
            ]
         }
      },
      "pwm": {
         "type": "array",
         "description": "PWM channels to configure.",
         "items": {
            "type": "object",
            "properties": {
               "id": {
                  "type": "string",
                  "description": "ID of the PWM.",
                  "title": "id"
               },
               "state": {
                  "type": "integer",
                  "description": "Enable/disable PWM channel.",
                  "title": "state",
```

```
                        "enum": [0,1]
                    }
                },
                "required": [
                "id",
                "state"
                ]
            }
        },
        "i2c": {
            "type": "array",
            "description": "I2C interfaces to configure.",
            "items": {
                "type": "object",
                "properties": {
                    "id": {
                        "type": "string",
                        "description": "ID of the I2C.",
                        "title": "id"
                    },
                    "rate": {
                        "type": "integer",
                        "description": "Transfer rate of the I2C channel.",
                        "title": "rate",
                        "minimum": 0,
                        "maximum": 2
                    }
                },
                "required": [
                "id",
                "rate"
                ]
            }
        },
        "uart": {
            "type": "array",
            "description": "UART interfaces to configure.",
            "items": {
                "type": "object",
                "properties": {
                    "id": {
                        "type": "string",
                        "description": "ID of the UART.",
                        "title": "id"
                    },
                    "baud": {
                        "type": "integer",
                        "description": "Transfer rate of the UART channel.",
                        "title": "baud",
                        "minimum": 0,
                        "maximum": 7
                    }
                },
                "required": [
                "id",
                "baud"
                ]
```

```
            }
         },
         "spi": {
            "type": "array",
            "description": "SPI interfaces to configure.",
            "items": {
               "type": "object",
               "properties": {
                  "id": {
                     "type": "string",
                     "description": "ID of the SPI.",
                     "title": "id"
                  },
                  "rate": {
                     "type": "integer",
                     "description": "Word size of the SPI channel.",
                     "title": "rate",
                     "minimum": 4,
                     "maximum": 16
                  }
               },
               "required": [
               "id",
               "rate"
               ]
            }
         },
         "can": {
            "type": "array",
            "description": "CAN interfaces to configure.",
            "items": {
               "type": "object",
               "properties": {
                  "id": {
                     "type": "string",
                     "description": "ID of the CAN.",
                     "title": "id"
                  },
                  "rate": {
                     "type": "integer",
                     "description": "Data rate of the CAN channel.",
                     "title": "rate",
                     "enum": [0,1,2]
                  },
                  "filter": {
                     "type": "array",
                     "description": "List of Filter Ids.",
                     "items": {
                        "type": "object",
                        "properties": {
                           "id": {
                              "type": "string",
                              "description": "ID of the CAN filter.",
                              "title": "id"
                           }
                        }
                     }
```

```
                    }
                },
                "required": [
                "id",
                "rate",
                "filter"
                ]
            }
        }
    }
}
```

Listing D.1: Basic JSON Schema [417] definition of the `configType` data type used in *Model-RPC*.

D.2 OpenRPC Schema Specification for the UART `write` Method

Listing D.2 provides an OpenRPC schema specification for the `uart.write` method of the *Model-RPC* communication protocol.

```
{
   "openrpc":"1.2.6",
   "info":{
      "title":"Model-RPC",
      "version":"2.0.0"
   },
   "methods":[
   {
      "name":"uart.write",
      "params":[
      {
         "name":"buffer",
         "description":"data string to be transferred to controller",
         "required":true,
         "schema":{
            "type":"string"
         }
      },
      {
         "name":"device",
         "description":"identifier of the UART controller",
         "required":true,
         "schema":{
            "$ref":"#/components/schemas/uartcontroller"
         }
      }
      ],
      "result":{
         "name":"Success",
         "schema":{
            "type":"boolean"
         }
      },
      "errors":[
      {
```

```
36              "code":-32602,
37              "message":"invalid method parameter(s)"
38           }
39           ]
40        }
41        ],
42        "components":{
43           "schemas":{
44              "uartcontroller":{
45                 "type":"object",
46                 "required":[
47                 "controller"
48                 ],
49                 "properties":{
50                    "controller":{
51                       "type":"integer",
52                       "minimum":0
53                    }
54                 }
55              }
56           }
57        }
58     }
```

Listing D.2: *Model-RPC* write method for UART specified with OpenRPC version 1.2.6 [283].

E Supplementary Information about Model-Testbeds

This section contains further information about the developed *Model-Testbeds* presented in Section 6.5 (p. 161 ff.). Section E.1 covers the mapping of power modes between the designed HAL (cf. Section 6.3, p. 149 ff.) and the *Model-Testbed* firmware, which has been adapted for a multi-MCU support. The schematics of the developed NXP LPC54114 breakout board are provided in Section E.2.

E.1 Power Modes

This section describes the power mode mapping for different MCU architectures used in this thesis for *Model-Testbeds*. The following Table E.1 shows the power mode specified by the HAL (left column of Table E.1) and implemented by the *Model-Testbed* firmware (cf. Section 6.5.2, p. 163 ff.) and the power modes provided by the used MCUs (middle column of Table E.1), namely the Espressif ESP32, NXP LPC54114, and STMicroelectronics STM32L476. The data in Table E.1 has been taken from data sheets [111, 269, 372] for each specific MCU.

Power Mode		**Description**
HAL	*MCU*	
	Espressif ESP32 [111]	
ACTIVE	Modem-sleep	The CPU is operational, and the clock is configurable. The Wi-Fi / Bluetooth baseband and radio are disabled. Expected power consumption: 44-27 mA (Dual Core), 34-27 mA (Single Core) @ 160 MHz.
SLEEP	Light-sleep	The CPU is paused. The RTC memory, RTC peripherals, and the ultra-low power co-processor are running. Any wake-up events, e.g., host, RTC timer, or external interrupts, will wake up the CPU. Expected power consumption: 0.8 mA.
DEEP_SLEEP	Deep-sleep	Only the RTC memory and RTC peripherals are powered. Wi-Fi and Bluetooth connection data are stored in the RTC memory. The ultra-low power co-processor is functional. Expected power consumption: 10 µA (RTC timer & RTC memory).
DEEP_POWER_DOWN	Hibernation	The internal 8 MHz oscillator and ultra-low co-processor are disabled. The RTC recovery memory is powered down. Only one RTC timer on the slow clock and certain RTC GPIOs are active. The RTC timer or the RTC GPIOs can wake up the CPU from Hibernation mode. Expected power consumption: 5 µA.
OFF	OFF	The `CHIP_PU` pin of the ESP32 is set to a low level; the CPU is powered off. Expected power consumption: 1 µA.
	NXP LPC54114 [269]	
ACTIVE	Active	The CPU is operational, and the clock is configurable. Expected power consumption: 9.9 mA with system clock @ 96 MHz (ARM Cortex-M4 core operates in active mode while the ARM Cortex-M0+ core is set to the sleep mode).
SLEEP	Sleep	The system clock (CPU) and instruction execution are stopped until reset or interrupt occurs. Peripheral functions may generate interrupts to cause the CPU to resume execution. The CPU state and registers, peripheral registers, and internal SRAM values are maintained, and the logic levels of the pins remain static. Expected power consumption: 3 mA with system clock @ 96 MHz.

Continued on next page

Table E.1 – continued from previous page

Power Mode		Description
HAL	*MCU*	
DEEP_SLEEP	Deep-sleep	The system clock to the CPU is disabled. Analog blocks, the main clock, and all peripheral clocks are powered down by default. The flash memory is put in standby mode. The CPU state and registers, peripheral registers, and internal SRAM values are maintained, and the logic levels of the pins remain static. GPIO Pin Interrupts, GPIO Group Interrupts, and selected peripherals such as USB, SPI, I^2C, UART, and RTC can be left running. The DMA may operate in deep-sleep mode. Expected power consumption: 16 µA with SRAM0–3 and SRAMX powered.
DEEP_PO–WER_DOWN	Deep power-down	Power is shut off to the entire chip except for the RTC power domain and the RESET pin. The MPU can wake up via the RESET pin and the RTC alarm. The contents of the SRAM and registers are not retained, and all functional pins are tri-stated. Expected power consumption: 290 nA, with the RTC oscillator disabled.
OFF	–	Not provided.
		STMicroelectronics STM32L476 [371]
ACTIVE	Run	CPU, DMA, and peripherals are active. Flash and SRAM are powered. Expected power consumption: 10.2 mA @ 80 MHz with source code and data processing running from Flash, external clock source (bypass mode), and phase-locked loop enabled.
SLEEP	Sleep	CPU is stopped. All peripherals continue to operate and can wake up the CPU when an interrupt or event occurs. Expected power consumption: 2.96 mA @ 80 MHz with external clock source (bypass mode), and phase-locked loop enabled and flash powered.
DEEP_SLEEP	Standby	The internal regulator, phase-locked loop, and high-speed external crystal oscillators are switched off. The RTC may remain active while the brown-out reset is permanently active. The MPU exits Standby mode when an external reset, an RTC event occurs, or a failure is detected on low-speed externals. Expected power consumption: 150 nA @ 3 V w/o independent and RTC disabled watchdog or 317 nA @ 3 V w/ independent watchdog and RTC disabled.
DEEP_PO–WER_DOWN	Shutdown w/ RTC	The internal regulator, phase-lock loop, and oscillators are switched off. The RTC remains active. No power voltage monitoring is possible. The mode can be exited by an external reset wake-up pin event or an RTC event (alarm, periodic wake-up, timestamp, tamper). Expected power consumption: 64.1 nA @ 3 V.
OFF	Shutdown w/o RTC	The internal regulator, phase-lock loop, oscillators, and RTC are switched off. No power voltage monitoring is possible. The mode can be exited by an external reset wake-up pin event or an RTC event (alarm, periodic wake-up, timestamp, tamper). Expected power consumption: 554 nA @ 3V with RTC clocked by low-speed external quartz in low drive mode.

Table E.1: Power Mode Mapping for the developed *Model-Testbeds*.

E.2 NXP LPC54114 Breakout Board Schematics

Figure E.8 pictures the schematic of the breakout board designed for the NXP LPC54114, while the pinout is shown in Figure E.9.

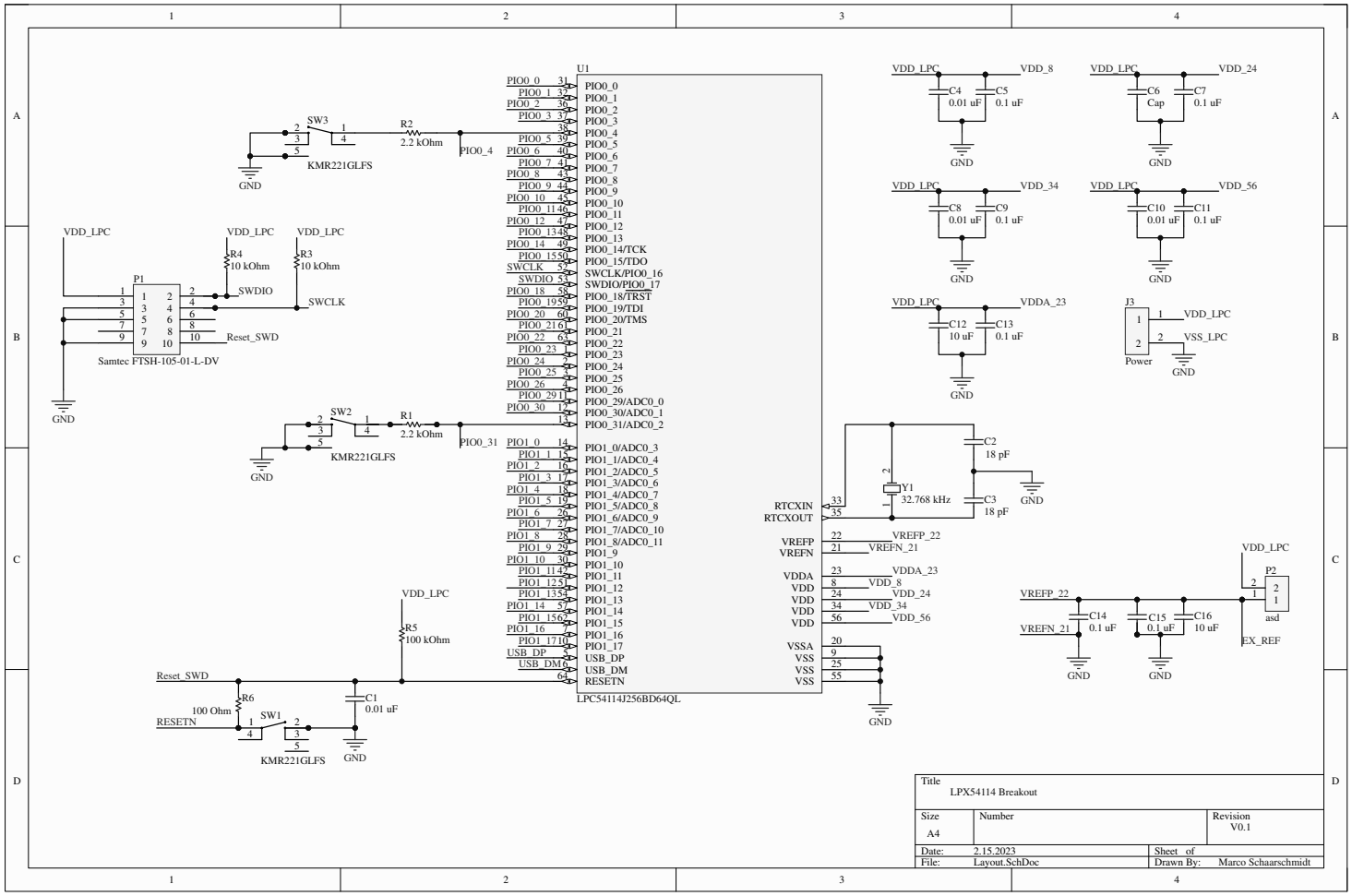


Figure E.8: Schematic of the NXP LPC54114 Breakout Board.

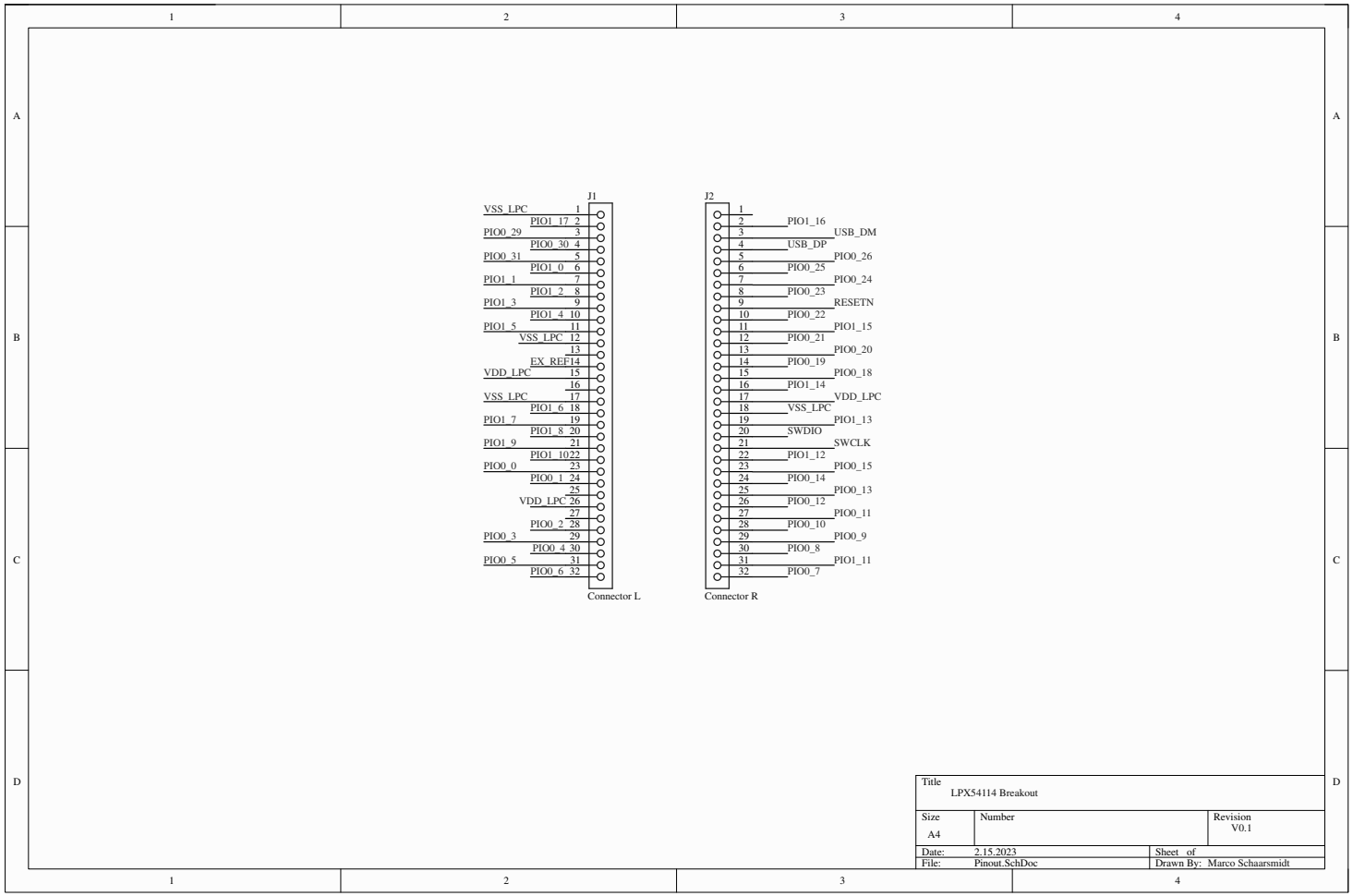


Figure E.9: Pinout of the NXP LPC54114 Breakout Board.

F Prototype Implementation Details

This section describes tools, technologies, and programming languages used for the proof-of-concept implementation and hardware platform details for evaluating the case study in Chapter 7 (p. 175 ff.).

Software

The UC^{2}E tool has been developed with Qt 5.15.0 [314] using C++17 and is built with optimization level `O3`. The Qoitech Otii Arc measuring device [313] has been updated to firmware version 1.1.6, while the measurements are obtained by the Otii desktop application in release 2.8.4. The firmware of the ESP32-based *Model-TestBed* has been developed using the Espressif IoT development framework version 4.3.1 [112]. As a UML modeling and simulation environment, IBM Rhapsody [164] in version 9.0.0 has been used.

Hardware

The host system executing IBM Rhapsody, the UC^{2}E tool, and the Otii desktop application is based on an Intel i5 6600 CPU, 16 GB RAM, 512 GB SSD, and is executing Windows 10 (Build 19044). The Qoitech Otii Arc measuring device offers a sample rate of 4 ksps for current measurements in a range of ±19 mA and 1 ksps for a ±2.7 A range with an accuracy of ±(0.1 % + 50 nA) and ±(0.1 % + 150 µA), respectively. For voltage measurement, the Otii Arc provides a sample rate of 1 ksps with an expected accuracy of ±(0.1 % + 1.5 mV).