

Power Consumption Estimation in Model Driven Software Development for Embedded Systems

Marco Schaarschmidt¹ ^a, Michael Uelschen¹ ^b and Elke Pulvermüller²

¹Faculty of Engineering and Computer Science, University of Applied Sciences Osnabrück, Germany

²Software Engineering Research Group, University of Osnabrück, Germany


Keywords: Model-Driven Development, Embedded Systems, UML, MARTE, Power Consumption, Energy Bug.


Abstract: Due to the resource-constrained nature of embedded systems, it is crucial to support the estimation of their power consumption as early in the development process as possible. Non-functional requirements based on power consumption directly impact the software design, e.g., watt-hour thresholds and expected lifetimes based on battery capacities. Even if software affects hardware behavior directly, these types of requirements are often overlooked by software developers because they are commonly associated with the hardware layer. Modern trends in software engineering such as Model-Driven Development (MDD) can be used in embedded software development to evaluate power consumption-based requirements in early design phases. However, power consumption aspects are currently not sufficiently considered in MDD approaches. In this paper, we present a model-driven approach using Unified Modeling Language profile extensions to model hardware components and their power characteristics. Software models are combined with hardware models to achieve a system-wide estimation, including peripheral devices, and to make the power-related impact in early design stages visible. By deriving energy profiles, we provide software developers with valuable feedback, which may be used to identify energy bugs and evaluate power consumption-related requirements. To demonstrate the potential of our approach, we use a sensor node example to evaluate our concept and to identify its energy bugs.

1 INTRODUCTION

Through the popularity of *Internet of Things* (IoT) and *Industrial Internet of Things* (IIoT), embedded systems are increasingly used in domains like environmental monitoring, smart cities, agriculture, and smart factories (Elijah et al., 2018; Abd El-Mawla et al., 2019; Zanella et al., 2014). According to (Gupta et al., 2017), the number of IoT-connected devices will reach 43 billion by the year 2023, which will represent 50 percent of all networked devices (Cisco Systems, 2020). The expected standby energy consumption of all IoT-connected devices will reach 46 TWh in 2025 (Friedli et al., 2016). As a result, *non-functional requirements* (NFRs) based on power consumption are gaining importance. This is especially true for battery-powered devices that are placed at inaccessible places (e.g., underground) (Vuran et al., 2018) or devices that do not have any energy harvesting capabilities. Due to the increasing complexity of algorithms and the size of

use cases, software applications of embedded systems are becoming more complex. Moreover, the variety of peripheral devices, processor architectures, operating systems, and communication interfaces leads to additional challenges in software development. Furthermore, the detection of so-called energy bugs is another important challenge for software developers in the field of embedded software development. In general, an energy bug (Banerjee et al., 2014) can be defined as a behavior of the complete system that causes an unexpected energy drain, which is not necessary to perform the actual functionality of the system. Typical energy bugs are caused by complex interactions between software and hardware components, peripheral misuse, incorrect use of APIs and drivers, flaws in the software design (e.g., preventing the system from entering a lower power state), and unoptimized or faulty source code (e.g., heavy usage of avoidable wait cycles) (Banerjee et al., 2014; Pathak et al., 2011). In contrast to conventional software bugs, energy bugs do not necessarily lead to misbehavior of the software itself and are not detectable without accurate simulations in early stages or detailed power consumption

^a  <https://orcid.org/0000-0001-8260-5326>

^b  <https://orcid.org/0000-0002-0841-6954>

measurements while the application is executed on a fully functional hardware platform in a laboratory setting. Because field or burn-in tests are not able to detect such application-based misbehavior (Silicon Labs, 2010), energy bugs have to be addressed during the early development stages.

For battery-powered devices, the energy consumption of software applications can be a significant bottleneck (Banerjee et al., 2016) and can cause up to 80 percent of the total energy consumption through software-hardware interactions (Georgiou et al., 2018). Software developers are often unaware of the impact of software on energy consumption (Pang et al., 2016) and it is therefore essential that NFRs related to power consumption are considered in early design phases, where changes are more effective (Tan et al., 2003). In the current software development of embedded systems, a power consumption analysis is typically carried out at the end of the development process. As a result, the potentially required re-design and optimization phases can result in time delays and increasing costs. Additionally, there exists no approach or tool support for a power estimation in early design phases, where the hardware platform may not be available or defined yet.

Model-Driven Development (MDD) is a widely accepted paradigm improving the correctness and efficiency in software engineering used both in research and industry (Domingo et al., 2020). By using multiple levels of abstraction, the overall complexity of software applications may be reduced to its essential complexity. With automatic software code generation processes, the quality of software applications can be increased. Since models are not bound to the underlying implementation by definition, they may help to overcome the aforementioned challenges during software development when different hardware architectures are used. If a specific software application is supposed to run on different hardware architectures, code adaptations for each target platform are necessary. With MDD, code generators can be used that can transform software models into source code for the specific target platform. In software development, the *Unified Modeling Language* (UML) (Object Management Group, 2017) is typically used to describe aspects like the general structure and behavior of the software application. Focusing on time behavior and schedulability, the *Modeling and Analysis of Real-time and Embedded systems* (MARTE) profile (Object Management Group, 2019a) extends UML to describe *non-functional properties* (NFP) of hardware and software and provides power consumption and dissipation modeling in a simplified way. However, the provided stereotypes are not sufficient to model dynamic power

consumption and power-related behavior in a granular way. Additionally, to the best of our knowledge, there exists no approach to link the software application model with hardware behavior models to obtain an early, rapid and straightforward power consumption estimation of a software application for given hardware configurations. To address the gap of power consumption estimation in MDD, we present the following, novel contributions in this paper:

- We address the gap between software and hardware modeling by describing hardware components with UML behavioral models. By linking these models, the impact of the software application on hardware components is made explicitly visible. Furthermore, by including the microcontroller, sensors, actuators, and communication interfaces, a system-wide power estimation can be achieved.
- We provide a UML-based power analysis profile to model power-related NFPs of hardware components. Our profile extends MARTE for a more fine-grained and dynamic characterization of hardware behavior when used from a software perspective.
- We propose a lightweight interchange format of hardware component models and power characteristics that enables *Model-2-Model* (M2M) transformations across development and analysis tools.
- Finally, we propose a workflow for software developers that describes the integration of our approach into the MDD process.

The general methodology of our proposed approach is shown in Figure 1. The approach aims to combine software models with hardware models to achieve an early estimation by using the concepts of MDD for power consumption analysis. We are using hardware

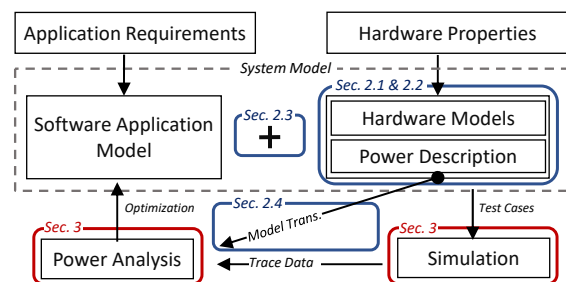


Figure 1: Overview and methodology of our approach.

properties to derive hardware models and extend those models with power consumption aspects. For every relevant hardware component from the software perspective, e.g., microcontroller, sensors, and actuators, a separate hardware model is defined. Thus, our approach provides a system-wide view and is not limited

to a specific group of hardware components. By using UML to describe hardware models, we are able to combine the software and hardware domain, which we define as a system model (c.f. Figure 1). The resulting model is completely integrated into the software domain and may be used to derive *energy profiles* when simulated. An *energy profile* describes the effect of the software application on power consumption over time for an embedded system when used with specific hardware components. This is an important step towards an energy transparent software application. By providing detailed feedback, the concept of *energy transparency* (Georgiou et al., 2018) allows software developers to make energy-aware decisions in order to meet power consumption requirements. Software developers may use *energy profiles* to visualize the energy-related impact of software applications, design changes, and algorithms. Moreover, *energy profiles* may be used to identify *energy bugs* and evaluate power consumption-based NFRs.

The remainder of this paper is organized as follows: Section 2 describes our approach, which is evaluated in Section 4 using an IoT-related use case. Section 3 introduces the integration of our approach into the development workflow. Related research and their conceptual differences are discussed in Section 5. Section 6 discusses our approach and Section 7 concludes this paper and presents future work.

2 RESEARCH APPROACH

This section describes our research approach for a model-based power consumption estimation of embedded systems. Figure 1 provides an overview of the proposed approach. It is divided into four main sections: The process of hardware abstraction (c.f. Section 2.1) and energy behavior modeling (c.f. Section 2.2), the integration of hardware models (c.f. Section 2.3), and M2M transformation (c.f. Section 2.4).

2.1 Hardware Component Modeling

A key challenge of our approach is to model the dynamic behavior of hardware components, extend those models with energy-related parameters and provide an interface to make the impact of the software application quantifiable. As a formal notation, we denote a hardware component model of an embedded system developed as H^{Sys} . Each hardware component model H_n^{Sys} is represented as a tuple $H_n^{Sys} = \{SM_n, OP_n, A_n\}$, where the elements of the tuple are finite sets defined as:

- SM_n represents a finite set of all states s , transitions

t and events e of the physical hardware component, so that $SM_n = \{s_n, t_n, e_n\}$. States s represent a list of operations modes, transitions t a list of possible changes between states in s , triggered by events e .

- OP_n represents a finite set of operations, which may be used by a software model, e.g., to change the configuration and trigger transitions.
- A_n is a set of attributes defining the inner state of the hardware component.

Several approaches, e.g., (Martinez et al., 2015; Zhu et al., 2014), analyze the dynamic behavior for different system states with varying levels of power consumption. As stated in (Zhou et al., 2011; Benini et al., 2000), each hardware component of an embedded system can be described with a set of states defining operating modes and transitions to switch between modes. The concept of *power state machines* (PSMs) (Benini et al., 2000; Danese et al., 2016) generally describes the annotation of states and transitions with meta-information related to power consumption. Our approach extends PSMs by modeling dynamic power characteristics for states and transitions depending on the current device configuration. We also map PSMs directly to UML *behavioral state machines* (BSMs) (Object Management Group, 2017), which can be used as a *classifier behavior* of UML class elements, representing hardware components. A class element C^{hw} is therefore suitable to represent hardware models while modeling software applications so that $C_n^{hw} = H_n^{Sys}$ when using UML. By this, software models can interact with hardware representations and simulate real hardware accesses, which is a crucial part of estimating power consumption (Georgiou et al., 2018).

2.2 Energy Behavior Modeling with the Power Analysis Profile

While MARTE is an expressive notation for modeling timing aspects, there exists only limited support for power-related characteristics and no accurate description of voltage and electrical current. We extended MARTE by adding those measurement units and corresponding NFP types as seen in Figure 2. The definition and description of the measurement units and the NFP types follow the MARTE specification (Object Management Group, 2019a) described in (Selic and Gérard, 2014). For the *ElectricCurrentUnitKind* in Figure 2, the symbol I describes the basic physical dimensions for electric current. *VoltageUnitKind* consists of the base dimension for mass (M), length (L), time (T) and electric current (I). For conversions, the *baseUnit* and *convFactor* are used. By specifying additional data types for voltage and electric current, it is

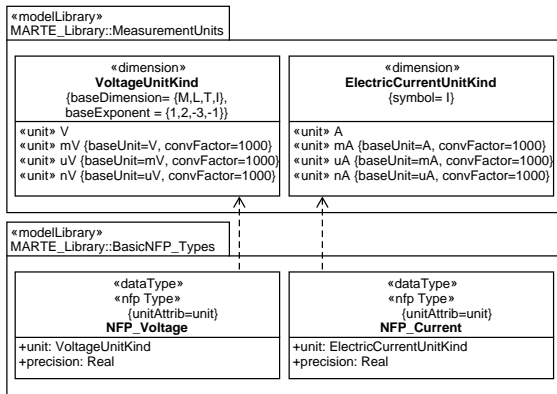


Figure 2: Extended Measurement Units and NFP Data Types.

possible to evaluate the power consumption of states and transitions in a dynamic and granulated manner.

To be able to model and analyze dynamic power characteristics, we define the *Power Analysis Profile* (PAP), which is based on MARTE and extends the profile by adding new tags and using the aforementioned units (e.g., *VoltageUnitKind*), and data types (e.g., *NFP_Voltage*) as shown in Figure 3. The pro-

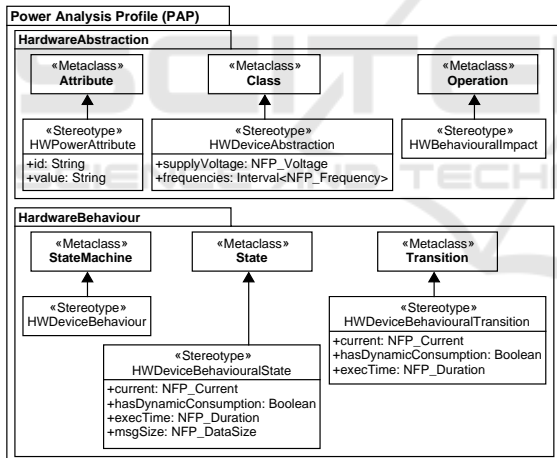


Figure 3: Simplified overview of the Power Analysis Profile.

vided stereotypes are divided into two main packages *HardwareAbstraction* and *HardwareDeviceBehaviour*. Stereotypes provided by the *HardwareAbstraction* package are specifically designed to describe abstract hardware components. UML classes annotated with *HardwareDeviceAbstraction* are used as a base representation of a hardware component model H^{Sys} . Due to the annotation, general properties can be specified such as the supply voltage and supported frequencies. In addition, operations and attributes influencing the behavior of a hardware component in terms of power consumption are annotated with *HWBehaviouralImpact* and *HWPowerAttribute*

respectively. The software can use those operations to change *HWPowerAttribute* annotated attributes, allowing dynamic behavior changes of the hardware component. By this, a connection between the software model and the hardware component model is defined.

Stereotypes provided by the *HardwareAbstraction* package can be used to express the power-related behavior. Each hardware model class C^{hw} includes a BSM defining operation modes and transitions. The *HWDeviceBehaviouralState* and *HWDeviceBehaviouralTransition* stereotypes are used to extend the BSM with power-related characteristics. For example, a software application changes the configuration (e.g., number of measurements) of a sensor during execution. To take this kind of dynamic behavior into account, our approach evaluates related parameters whenever a state is entered or transition is executed.

In general, the *Value Specification Language* (VSL) (Object Management Group, 2019a; Selic and Gérard, 2014) is used to model dynamic behavior. However, the basic concept was slightly adapted to be able to express the relation between tags of states or transitions and tags and attributes from the instance of a class. When *NFP types* from the MARTE specification are used for tagged values, they can be expressed as a tuple (*value, expr, unit, source, precision, statQ, dir*) (Object Management Group, 2019a). In this paper, the first three elements are used. The *value* element contains the actual value and can be expressed as numerical quantity or string. *expr* is an optional element and contains a *VSL_Expression*, if an expression instead of a fixed value is used and *unit* specifies the physical measurement unit. As basic example for the tuple notation, the tagged value *current* (c.f. Figure 3) can be expressed as (*value = 1.5, unit = mA*), (*1.5, -, mA, -, -, -, -*) or (*1.5, mA*) if the shortened notation is used.

In related approaches (c.f. Section 5), a fixed numeric term is used to describe the power, execution time, and other tagged values within a state and a transition. Our approach presented in this paper does not have this limitation and is able to model a dynamic behavior by using expressions, which can be evaluated dynamically during the analysis process when a state is entered or a state transition is executed. By modeling dynamic behavior, it is now possible to include modifiable configurations of hardware components and to evaluate the interaction of these components with the software application model during simulation. A typical example of a configuration is the number of repeated measurements of a sensor before an average value is calculated. If the configuration of a

hardware component is changed by the software application during simulation, the power consumption for a specific state or transition may be affected and can be re-evaluated dynamically. A change in the behavior can also occur if the software application varies the amount of data to be transmitted by a communication interface, which affects the time the hardware component model stays in a transmission state. A dynamic behavior can be declared by setting the value of the tag *hasDynamicConsumption = true*. If set, tagged values based on NFP types containing an expression. In general, the expression consists of elements from the sets V_n , C_n and O_n , where

- V_n denotes all variables leading to a dynamic behavior for a given state or transition of a hardware component.
- C_n represents a list of constants.
- O_n denotes a finite set of mathematical operators to combine constants and variables.

The definition of variables in V_n is based on the VSL and uses the *Variables* type from the *VSL::Expressions* package. In our approach, the definition and usage are slightly adapted and differ from the specification to enable cross-references between tags of different states and transitions as well as attributes from the instance of a class C^{hw} to achieve a dynamic behavior. First, the MARTE specification stated a 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). In our approach, the evaluation has to be performed whenever the value of variables changes during simulation. Second, the *VSL::Expressions::Variables* used in this approach are not declared explicitly. Instead, we use a specific naming scheme to achieve a linkage between the variable definition and the tagged value or attribute it is linked to. For the process of power consumption optimization we defined the following schemes for variables:

- $\$ \ll tag \gg$: Denotes the tagged value for a tag in the scope of the current state or current transition.
- $\$SM. \ll NameOfState \gg . \ll tag \gg$: Represents a tag of a state within the same BSM.
- $\$SM. \ll NameOfTransition \gg . \ll tag \gg$: Represents a tag for a transition within the same BSM.
- $\$ATTR. \ll attributeId \gg$: Denotes an attribute of the hardware abstraction class H^{Sys} annotated with the *HWPowerAttribute* stereotype. The *id* tag must match $\ll attributeId \gg$.

An example for a dynamic execution time (*execTime*) can be expressed as ($expr = 2.0 \cdot \$ATTR.var, unit =$

ms). If the current consumption of two states *A* and *B* are linearly dependent, the tagged value for state *B* can be expressed as ($expr = \$SM.A.current \cdot 2, unit = mA$). In the following chapters, the shortened notation ($value|expr, unit$) is used for a better readability.

2.3 Combining Software & Hardware Models

To achieve a power consumption estimation in early design phases, software developers have to integrate hardware models into the software model domain, allowing a software model to be tested even when the hardware platform is not fully finalized. Furthermore, different types of hardware models can be integrated and thus performance comparisons or design-space explorations (Gries, 2004) can be performed. Figure 4 shows the representation and integration of hardware models into a software application model. The *Hardware Manager* class in Figure 4 provides a centralized management and monitoring of all hardware models of a system that are essential for the software application. Therefore, it serves as a central interface between the software and hardware models. All hard-

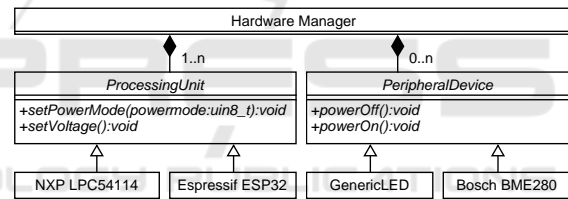


Figure 4: Integration of abstract hardware components.

ware models are assigned to one of the two abstract device classes (c.f. Figure 4), representing *Processing Units* and *Peripheral Devices* like sensors, actors, and communication interfaces. Thereby, some basic power-related operations are provided, which should be implemented by a hardware model to provide an interface for the software application. Each generalization should follow the *Hardware Proxy Pattern* (Douglass, 2011) representing peripheral devices in the software model domain and include an interface for the software model and a state machine to express power-related behavior. Interface descriptions can be derived, e.g., from existing driver descriptions. This has an advantage for the following code generation by MDD tools because hardware models can be replaced with driver implementations while leaving the software application unchanged. By this, the generated software application remains platform-independent. The implementation of the processor is a special challenge due to its complexity, which requires further abstraction. There exists a great difference in the characteristics and number of operat-

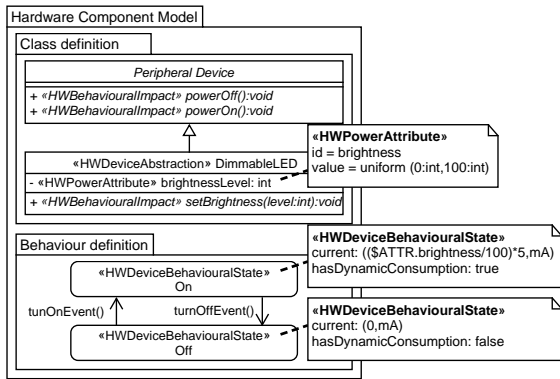


Figure 5: Basic example of the applied research approach.

ing modes of processors. For example, each processor family has a different number of operating modes and implements a different procedure for powering the flash, SRAM banks, or oscillators in individual operating modes. To be able to provide a consistent interface for the software model, the abstract class *ProcessingUnit* provides a set of predefined *PowerModes* = {ACTIVE, SLEEP, DEEP_SLEEP, DEEP_POWER_DOWN, OFF} which has to be mapped to the existing power modes of the specific microcontroller. When the software application has to be generated for different processor architectures, a hardware abstraction layer (HAL) approach has to be implemented and linked before the compilation.

Figure 5 shows a basic hardware model example of a *DimmableLED* using the aforementioned concepts and techniques. The class definition (c.f. the upper part of Figure 5) is derived from the base class *Peripheral Device* (c.f. Figure 4) and is extended by adding an internal attribute *brightnessLevel*, describing the current brightness of the LED as well as a method to change the attribute by the software model. The stereotype *HWDeviceAbstraction* is used to set a unique id as well as a specification about the value range. For example, this property may be used by simulation environments when return values from sensors or radio interfaces have to be generated. When the functions *powerOff()* and *powerOn()* are called, events are emitted initiating state transitions. The behavior is represented by a state machine (lower part of Figure 5). The state machine consists of two states *Off* and *On*, which are extended with the *HWDeviceBehaviouralState* stereotype from the PAP. The tagged value of the *current* tag in the state *Off* is set to a fixed value, while an expression is used to define a dynamic current consumption in the state *On*. Depending on the attribute *brightness* of the *DimmableLED* class, the current consumption can vary between 0.05 mA and 5 mA. If the software model changes the brightness level during simulation, the tagged value will be updated with the new value.

2.4 Model-2-Model Interchange Format

M2M transformations are an important aspect of MDD. In our approach, a M2M transformation of hardware component models is used along with trace analysis to perform a power consumption estimation with external tools like MATLAB (The MathWorks, Inc., 2021). We provide a lightweight JSON-based (Nurseitov et al., 2009) interchange format for *Meta Object Facility* (MOF) (Object Management Group, 2019b) based models, which are annotated with our PAP. During the creation of the JSON-based description, all UML model elements are processed and searched for PAP annotations. Beside the basic structure of state machines (e.g. states and transitions), power-related tagged values and class attributes (e.g., names, data types, ranges) are also transformed to take the dynamic behavior into account. This information is subsequently stored in a JSON file. External tools like MATLAB can import the hardware component model descriptions to be aware of the energy-related dynamic behavior as well as the expected power consumption based on the actual hardware states. After the simulation, traces can be used for the power consumption estimation process. The M2M approach is part of the proof of concept for the analysis process based on models annotated with PAP. Figure 6 shows a basic example of the JSON-based description containing multiple states, transitions, and class attributes.

```

1  "Statemachine": [{
2    "Name": "BME280",
3    "Attr": {
4      "s_oversampling": {
5        "id": "S_Sample",
6        "value": "uniform(0.0,16.0)"
7      },
8      [...]
9    },
10   "States": {
11     "FORCED": {
12       "Id": "6917cdb7-f8a3-4c78-835b-2150a3da38f6",
13       "Behaviour":{
14         "consumption": "(((1000/$execTime)/1000[...], uA)",
15         "execTime": "(1+(2*$ATTR.T_Sample)[...],ms)",
16         "hasDynamicConsumption": true
17       }
18     },
19     [...]
20   },
21   "Transactions": {
22     "1": {
23       "DefaultTransition": false,
24       "FromState": "ed4869b3-5e1b-4dfd-8ce8-56f1adda58e3",
25       "ToState": "8e44f019-e954-4420-a3d6-e2e3d660ac10",
26       "Behaviour":{
27         "consumption": "(0, uA)",
28         "execTime": "(0, ms)",
29         "hasDynamicConsumption": false
30       },
31     },
32     [...]
33   }
34 }

```

Figure 6: Basic example of a JSON based BSM description.

3 EVALUATION

In this section, we evaluate our approach using the methodology described in Figure 1 (introduced in Section 1) to obtain an energy profile of a software application for a sensor node in an early design phase. We use IBM Rational Rhapsody 8.4 (IBM, 2021) as an MDD tool and simulation environment for software and hardware models. To evaluate the system in terms

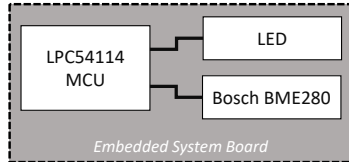


Figure 7: Hardware components used for this use case.

of power consumption, we define an exemplary hardware platform using an NXP LPC54114 low-power microcontroller (NXP Semiconductors, 2019) and two peripheral devices, as shown in Figure 7. The NXP LPC54114 is based on an ARM Cortex-M4 and also includes an ARM Cortex-M0+ co-processor, which has been disabled for this use case. The first peripheral device is a Bosch BME280 (Bosch Sensortec GmbH, 2018) environmental sensor for measuring temperature, barometric pressure, and humidity. A LED as a second peripheral device is used as a visual output. All hardware models are derived from the corresponding data sheets (c.f. (Bosch Sensortec GmbH, 2018), (NXP Semiconductors, 2019)). The behavior of the LED can be described with two basic states *On* and *Off*, annotated with a static current of 10 mA for the state *On* and 0 mA for the state *Off*. The behavior of the BME280 is more complex, resulting in a more extensive BSM as shown in Figure 8. When powered,

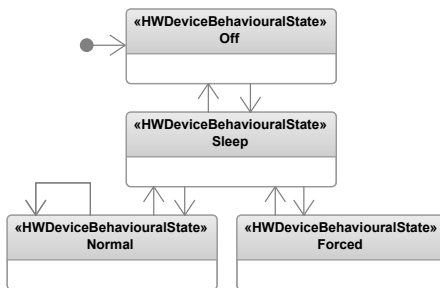


Figure 8: BSM of the Bosch BME280.

the sensor is operating in the mode *Sleep*. The sensor can be dynamically configured to take a single measurement (Mode: *Forced*) or to continuously take measurements (Mode: *Normal*). The oversampling rate for each sensor (e.g., temperature, pressure, humidity) can be configured individually. Therefore, an attribute for each sensor was added to the class definition

of the BMW280 using the *HWPowerAttribute* stereotype, while using the names *T_Sample*, *P_Sample*, and *H_Sample* respectively as tagged values for the tag *id*. Additionally, another attribute denoting the interval between measurements in *Normal* mode has been added. A software application may dynamically configure the oversampling rate of each sensor (e.g., temperature, pressure, humidity) individually, influencing the amount of electric current and the measurement time. To address this, the tag *id* of the attributes are referenced in the value fields of the *current* and *execTime* tags from stereotypes provided by the *HardwareBehaviour* package. Figure 9 shows the usage of such references in a more complex example for the state *Forced*. The amount of electric current, as well as the execution time for this state, depends on the current configuration of the sensor. It is important to notice, that a change in one of the defined attributes annotated with the *HWPowerAttribute* stereotype will influence the amount of electric current and the execution time for the state as shown in Figure 9. As a realization of the *Hardware Proxy Pattern* for the software-hardware interaction between the BME280 hardware model and the software application model, we abstracted the existing API definition (Bosch Sensortec GmbH, 2021) and applied the *HWBehaviouralImpact* stereotype on each operation affecting the power-related behavior. The state machine of the LPC54114 is shown in Figure 10. The tagged values for *current* in each state of the microcontroller are static and taken from direct measurements and data sheets. Since the execution of the software application is responsible for all state changes, the tag *execution time* is left empty and is not needed. In this scenario only the state *Active* and *Sleep* are used. For those states, the current is set to 9.9 mA and 3 mA respectively. We implement an executable software application model, as shown in Figure 11. As a typical use case for smaller IoT systems, a software implementation was defined, measuring and evaluating temperature values. The implementation consists of a single class containing a BSM with four different working states. The four defined states of the BSM

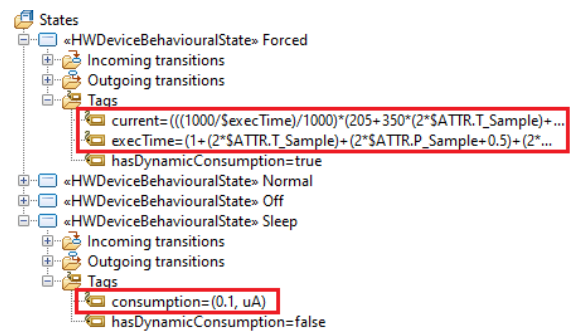


Figure 9: BME280 with annotations in Rational Rhapsody.

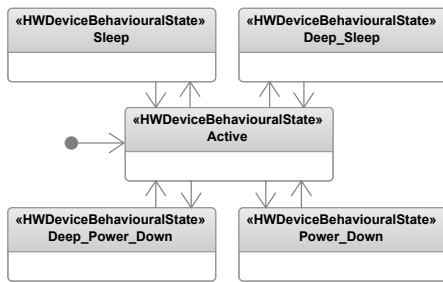


Figure 10: BSM of the NXP LPC54114.

describe the following actions:

- *Input*: Represents the part of the software application where a single measurement with the BME280 (*Forced* mode) is performed.
- *Process*: In this state, the measurement is processed. If a threshold is exceeded, the application will change to the *Output* state and to *Sleep* otherwise.
- *Output*: In this state, the LED will be enabled as a visual output and the system will switch to the *Sleep* state afterwards.
- *Sleep*: Denotes the state in which the LPC54114 is set to a low power mode for a fixed amount of time before re-entering the *Input* state automatically.

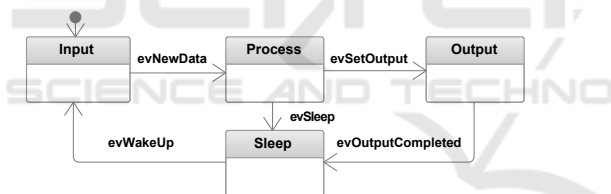


Figure 11: Software Model.

As preparation for the simulation, a measurement series with BME280 was performed. The measured values are passed back to the software model when the hardware model of the sensor is queried. During the simulation, trace logs of software-hardware interactions and hardware model state changes were recorded and used along with a M2M transformation of the hardware models for a trace analysis in MATLAB. Figure 12 presents the resulting energy profiles of the software application model interacting with the defined hardware components models in two different scenarios. The power estimation and accumulated energy consumption in the upper part of Figure 12 describes a scenario containing an energy bug preventing the NXP LPC54114 to enter a lower power state while the LED is turned on because of a bug in the transition between the *Output* and the *Sleep* state of the software application model (c.f. Figure 11). The lower part of Figure 12 describes the energy profile after the

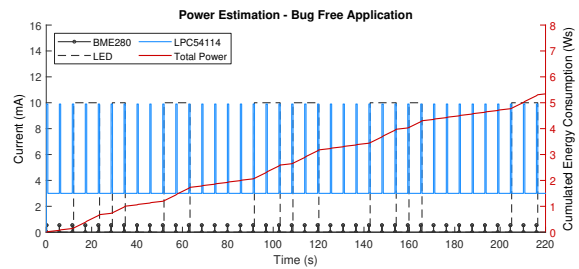
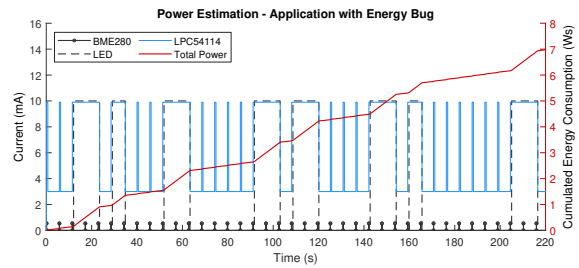


Figure 12: Analysis of the behavior with MATLAB.

software developer fixed the energy bug (e.g. optimization of the software model) and re-runs the simulation. In this example, the simulation was executed for a total of 220 seconds. The total consumption could be reduced from approx. 6.99 Ws to 5.38 Ws after the energy bug has been fixed. In general, it is advisable to use energy design patterns during the development of the software application (Schaarschmidt et al., 2020). Our approach may also be used to evaluate the impact of such design patterns on the entire system. This illustrates that our approach is able to analyze software-hardware interactions, optimize software applications, and evaluate power consumption-based NFRs like the duration until an energy source may be exhausted. The presented approach can also be used to evaluate subsystems related to both, hardware and software to evaluate individual program sequences or test cases.

4 DEVELOPMENT WORKFLOW

While the previous section evaluates our approach with an IoT-related use case, this section discusses a corresponding integration of the approach into a MDD development workflow. Figure 13 shows a UML activity diagram of the proposed workflow.

In the first step, a requirement list is used to define functional and non-functional requirements of the software application model. Alongside the development of the software application model in step 2, relevant hardware components are identified by the software developer (step 3). Only those hardware components are taken into account that can be directly influenced by the software model, e.g., sensors, actuators, communi-

cation interfaces, and microcontrollers. The software developer may use a model library from an internal or external, private or community-driven service providing hardware component model descriptions based on the presented M2M interchange format (c.f. Section 2.4), which can be directly imported into modern MDD tools, such as IBM Rational Rhapsody or Enterprise Architect (SparxSystems, 2020). The model library can be hosted, for example, by a centralized organization or by hardware vendors who provide drivers/APIs, data sheets, and hardware models as a combined package.

If no hardware component model description is available, the software developer can derive a hardware model by following steps 4(a) to 4(f). The state machine is based on the working or power modes of the hardware component and can be derived from the data sheet. As a next step, stereotypes from the *HardwareAbstraction* packages of the PAP can be used to add energy and time properties to states and transitions (steps 4(b)-4(c)). Afterward, the UML class is defined, which represents the interface for the software model and contains attributes, which can be used to describe the energy and time behavior in a precise manner. To declare functions and attributes that affect the behavior of hardware component models, stereotypes from the *HardwareAbstraction* packages are added. In step 4(f) the model is exported and stored in the model library.

When the modeling process of the software model and hardware components is finalized, a M2M transformation is performed. For each UML element that has been extended with at least one stereotype provided by PAP, a corresponding trace functionality is generated. Furthermore, a *Hardware Manager* is generated, so that the software model and hardware component models can be used together in the same simulation environment of a MDD tool. The *Hardware Manager* also aggregates traces for external power analysis tools. The result is an intermediate model which is able to provide a socket connection and trace logs when simulated so that external power analysis tools are able to analyze behavior changes of all hardware components.

In step 7, both models are linked together by including function calls of the hardware model class definition in the software model. When source code is generated, classes from hardware components can be replaced by driver implementations. Since the function signatures of the hardware model and the driver interfaces are identical, no code adjustments are required in the generated software application. A simulation of the software application model is performed in step 8. Afterward, the software developer may check whether the requirements are met or optimize the soft-

ware model and repeat the simulation. The steps 4(f), 5(a), and 6 shown in Figure 13 can be executed automatically. Depending on the toolchain used, the simulation and analysis process can be automatized. As a proof of concept, we implemented a plugin for IBM Rational Rhapsody which is able to exchange hardware models based on our proposed M2M interchange format (c.f. Section. 2.4) and is also capable of generating state machines and UML classes. Furthermore, the M2M transformation process to generate the *intermediate model* has been automatized. Both approaches were used for the evaluation in the previous Section 4.

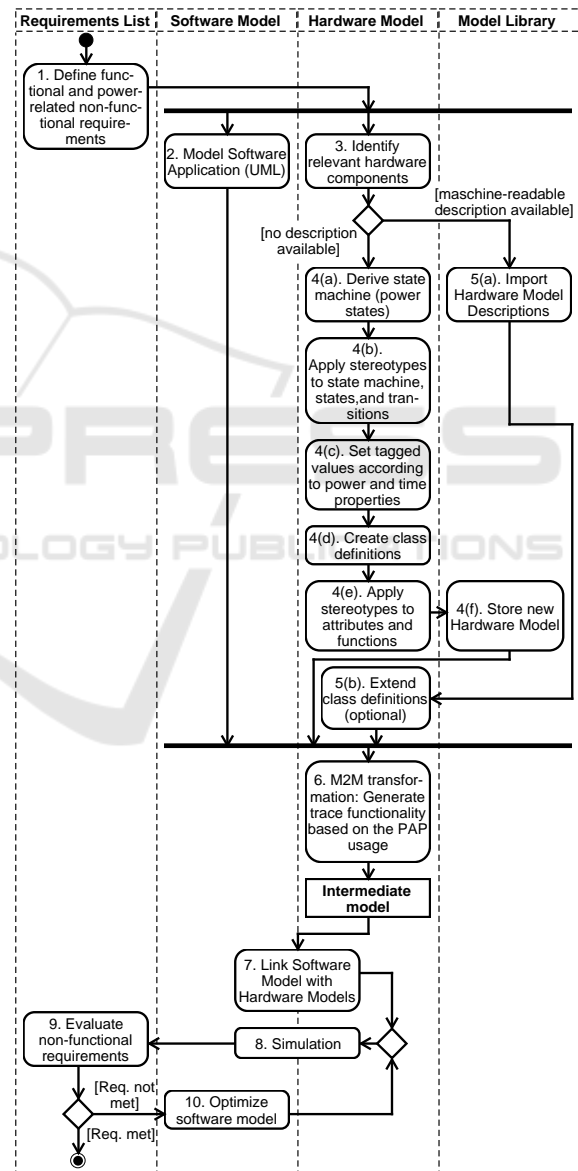


Figure 13: Workflow for the proposed approach.

5 RELATED WORK

Concepts for modeling NFRs like power consumption or execution time have been proposed in a set of research approaches using mathematical models (Martinez et al., 2015; Bouguera et al., 2018; Zhou et al., 2011), workflow models (Zhu et al., 2014), and Petri nets (Andrade et al., 2010). However, they do not consider the integration into software development and MDD.

Low-level approaches such as cycle-level or instruction-level power analyses (ILPA) are less suitable for evaluating software application models in early phases since they require a high level of knowledge about architectures and also have lengthy simulation times (Julien et al., 2003). ILPA for example requires assembly code for estimating the cost of each instruction and is therefore not close to the software model. Since these approaches are used primarily for processor simulations, they are also not suitable for a system-wide view that includes peripheral devices.

The authors in (Atitallah et al., 2015) provide a power estimation solution and low-level device modeling based on BSM and an extension of the IP-XACT standard (IEEE SA, 2014) to define power-related properties. Since their work focuses on register level for lower-level hardware components like clock generators, it is not suitable for analysis in early design phases in the MDD domain. Furthermore, their approach does not consider peripheral devices and their dynamic behavior.

There exist several approaches that are using UML and MARTE to model software and hardware aspects for a power consumption estimation of software applications for embedded systems. In (Gomez et al., 2012), a multi-view power modeling approach based on UML, SysML, and MARTE is proposed to model power aspects for different functional and structural system views. The authors strongly focus on connections between views and do not consider a power analysis of single components nor the impact of software applications.

In (Iyengar and Pulvermueller, 2018), a model-driven energy-aware timing analysis is presented, where the design model is obtained by reverse engineering and is mapped to UML classes and operations. MARTE is used to annotate the software model with timing and power-related properties. For analysis, a M2M transformation process from UML to a timing-energy analysis model is used, where classes are mapped to tasks and operations to runnable representations. Since the model is derived from a reverse engineering process, this approach is not suitable for an analysis in early design phases. Furthermore, only

the processor instead of the complete system, including peripheral devices, is considered.

In recent works, extensions for MARTE improving concepts for modeling and analyzing power characteristics are proposed. In (Hagner et al., 2011), a MARTE-based power consumption analysis view profile is introduced, defining new stereotypes to specify power-related characteristics like the processor's switching capacitance or energy consumption during the execution of tasks. Processors are annotated with voltage and frequency parameters and tasks with the number of cycles to find an optimal power solution when simulating a dynamic voltage scaling.

A MARTE profile extension for dynamic power management is proposed in (Arpinen et al., 2012). The authors are using BSMs annotated with power specifications to represent operation modes of each considered hardware component. States of hardware components are combined to system power configurations. As long as a configuration is active, the components remain in their state. Use cases allocating configurations are defined to generate and analyze workflows.

However, the aforementioned approaches focus on hardware-centric views when modeling and analyzing power consumption, while only using tasks and predefined use cases to describe software aspects. By this, those approaches barely support software developers in the task of quantifying the influence of a software model on power consumption. An estimation based on abstract use cases or at task level, as suggested by the previous approaches, is not sufficient and does not offer any possibilities for software optimizations. In contrast, our approach is integrated into the development process, focusing on software execution and software-hardware interaction.

6 DISCUSSION

Our approach enables software developers to estimate power consumption in early design phases. By linking software application models with hardware model representations, we are able to make software-hardware interactions directly visible within the model. Our novel UML-based profile enables a fine-grained and dynamic characterization of hardware-behavior and takes hardware-software interactions into account. A trace analysis may be used to derive energy profiles and identify energy bugs of software application models in early design phases. Additionally, design-space exploration methods (Gries, 2004) may be performed to determine the best hardware-software configuration based on energy consumption. NFRs like total peak power or battery capacity may be evaluated, result-

ing in design changes in the application's workflow. The visualization of possible energy bugs also helps to improve the quality of the application regarding power consumption. Our approach is tool independent and with the lightweight interchange format, hardware models may be exchanged between tools and reused in different projects, which increases developer productivity. We also presented a development workflow describing the integration of our approach into a MDD workflow from a software developer perspective.

Our approach also faces limitations. It is the nature of modeling, that accuracy decreases with abstraction. Parameters modeled with PAP are derived from data sheets or have been previously measured in specific environments (e.g., temperature), resulting in a loss of accuracy. Another limitation is caused by simulation environments of MDD tools where the runtime does not match the runtime of generated code directly executed on an embedded system. Overall, we recommend our approach, as it offers valuable feedback for developers. Additionally, an energy-related re-design and optimization of the software application may be performed in early design phases, reducing the overall development time and costs.

7 CONCLUSION

In this paper, we presented an approach to estimate the power consumption of software applications for embedded system development in early design phases without the need for real hardware components. Based on MARTE, we created hardware component models and included dynamic power characteristics in their descriptions. Our concept offers a novel approach to provide power estimations for software applications in embedded system environments by considering software-hardware interactions and making the dynamic power-related behavior in hardware states quantifiable. During simulations, hardware accesses can be traced, analyzed, and evaluated. Besides, our approach may be used by software developers for the detection of some typical energy bugs in early design phases. The early analysis and correction of these types of bugs lead to an improved quality of the application.

Future work includes the support for energy sources and communication interfaces. As a powerful toolset for creating energy profiles and evaluating NFR, we are also planning to provide a simulation and analysis environment. Furthermore, we want to compare our approach with physical measurements on real hardware platforms and evaluate our approach using more complex use cases.

REFERENCES

- Abd El-Mawla, N., Badawy, M., and Arafat, H. (2019). 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.
- Andrade, E., Maciel, P., Falcão, T., Nogueira, B., Araujo, C., and Callou, G. (2010). Performance and energy consumption estimation for commercial off-the-shelf component system design. *Innovations in Systems and Software Engineering*, 6(1-2):107–114.
- Arpinen, T., Salminen, E., Hämäläinen, T. D., and Hännikäinen, M. (2012). Marte profile extension for modeling dynamic power management of embedded systems. *J. Syst. Archit.*, 58(5):209–219.
- Atitallah, Y. B., Mottin, J., Hili, N., Ducroux, T., and Godet-Bar, G. (2015). A power consumption estimation approach for embedded software design using trace analysis. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 61–68, Piscataway, NJ. IEEE.
- Banerjee, A., Chattopadhyay, S., and Roychoudhury, A. (2016). On testing embedded software. In *Advances in Computers*, volume 101, pages 121–153. Elsevier.
- Banerjee, A., Chong, L. K., Chattopadhyay, S., and Roychoudhury, A. (2014). Detecting energy bugs and hotspots in mobile apps. *FSE 2014*, page 588–598, New York, NY, USA. Association for Computing Machinery.
- Benini, L., Bogliolo, A., and de Micheli, G. (2000). A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316.
- Bosch Sensortec GmbH (2018). BME280 – Data sheet, Version 1.6. Document Number BST-BME280-DS002-15 (<https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf>).
- Bosch Sensortec GmbH (2021). Github: BME280 sensor API. <https://github.com/BoschSensortec/BME280.driver>. Accessed: 12.02.2021.
- Bouguera, T., Diouris, J.-F., Chaillout, J.-J., Jaouadi, R., and Andrieux, G. (2018). Energy consumption model for sensor nodes based on lora and lorawan. *Sensors*, 18(7):2104.
- Cisco Systems (2020). Cisco annual internet report (2018–2023). White Paper C11-741490-01 (<https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>).
- Danese, A., Pravadelli, G., and Zandonà, I. (2016). Automatic generation of power state machines through dynamic mining of temporal assertions. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 606–611.
- Domingo, Á., Echeverría, J., Pastor, Ó., and Cetina, C. (2020). Evaluating the benefits of model-driven development. In Dustdar, S., Yu, E., Salinesi, C., Rieu, D., and Pant, V., editors, *Advanced Information Systems*

- Engineering*, pages 353–367, Cham. Springer International Publishing.
- Douglass, B. P. (2011). *Design patterns for embedded systems in C: An embedded software engineering toolkit*. Newnes/Elsevier, Oxford and Burlington, MA.
- Elijah, O., Rahman, T. A., Orikumhi, I., Leow, C. Y., and Hindia, M. N. (2018). An overview of internet of things (iot) and data analytics in agriculture: Benefits and challenges. *IEEE Internet of Things Journal*, 5(5):3758–3773.
- Friedli, M., Kaufmann, L., Paganini, F., and Kyburz, R. (2016). Energy efficiency of the internet of things: Technology and energy assessment report prepared for iea 4e edna.
- Georgiou, K., Xavier-de Souza, S., and Eder, K. (2018). The iot energy challenge: A software perspective. *IEEE Embedded Systems Letters*, 10(3):53–56.
- Gomez, C., DeAntoni, J., and Mallet, F. (2012). Multi-view power modeling based on uml, marte and sysml. In Cortellessa, V., editor, *38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), 2012*, pages 17–20, Piscataway, NJ. IEEE.
- Gries, M. (2004). Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38(2):131–183.
- Gupta, A., Tsai, T., Rueb, D., Yamaji, M., and Middleton, P. (2017). Forecast: Internet of things: Endpoints and associated services, worldwide, 2017.
- Hagner, M., Aniculaesei, A., and Goltz, U. (2011). Uml-based analysis of power consumption for real-time embedded systems. In *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1196–1201. IEEE.
- IBM (2021). Rational Rhapsody Developer. <https://www.ibm.com/products/uml-tools>. Accessed: 12.02.2021.
- IEEE SA (2014). IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. Document Number IEEE 1685-2014 (<https://standards.ieee.org/standard/1685-2014.html>).
- Iyengar, P. and Pulvermueller, E. (2018). A model-driven workflow for energy-aware scheduling analysis of iot-enabled use cases. *IEEE Internet of Things Journal*, 5(6):4914–4925.
- Julien, N., Laurent, J., Senn, E., and Martin, E. (2003). Power consumption modeling and characterization of the ti c6201. *IEEE Micro*, 23(5):40–49.
- Martinez, B., Monton, M., Vilajosana, I., and Prades, J. D. (2015). The power of models: Modeling power consumption for iot devices. *IEEE Sensors Journal*, 15(10):5777–5789.
- Nurseitov, N., Paulson, M., Reynolds, R., and Izurieta, C. (2009). Comparison of json and xml data interchange formats: a case study. *Caine*, 9:157–162.
- NXP Semiconductors (2019). LPC5411x - Product data sheet, Rev. 2.5. Document identifier LPC5411x (<https://www.nxp.com/docs/en/data-sheet/LPC5411X.pdf>).
- Object Management Group (2017). Unified Modeling Language, Version 2.5.1. OMG Document Number formal/17-12-05 (<https://www.omg.org/spec/UML/2.5.1/>).
- Object Management Group (2019a). 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/>).
- Object Management Group (2019b). Meta Object Facility, Version 2.5.1. OMG Document Number formal/19-10-01 (<https://www.omg.org/spec/MOF/2.5.1/>).
- Pang, C., Hindle, A., Adams, B., and Hassan, A. E. (2016). What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89.
- Pathak, A., Hu, Y. C., and Zhang, M. (2011). Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. HotNets-X, New York, NY, USA. Association for Computing Machinery.
- Schaarschmidt, M., Uelschen, M., Pulvermüller, E., and Westerkamp, C. (2020). 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 - Volume 1: ENASE*, pages 62–73.
- Selic, B. and Gérard, S. (2014). *Modeling and analysis of real-time and embedded systems with UML and MARTE: Developing cyber-physical systems*. Morgan Kaufmann, Waltham, MA.
- Silicon Labs (2010). Energy debugging tools for embedded applications. Technical report.
- SparxSystems (2020). Enterprise architect tool. <https://sparxsystems.com/products/ea/index.html>. Accessed: 12.02.2020.
- Tan, T. K., Raghunathan, A., and Jha, N. K. (2003). Software architectural transformations: a new approach to low energy embedded software. In *Design, Automation, and Test in Europe Conference and Exhibition*, pages 1046–1051, Los Alamitos, CA. IEEE Computer Society.
- The MathWorks, Inc. (2021). MATLAB. <https://www.mathworks.com/products/matlab>. Accessed: 12.02.2021.
- Vuran, M. C., Salam, A., Wong, R., and Irmak, S. (2018). Internet of underground things in precision agriculture: Architecture and technology aspects. *Ad Hoc Networks*, 81:160–173.
- Zanella, A., Bui, N., Castellani, A., Vangelista, L., and Zorzi, M. (2014). Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32.
- Zhou, H.-Y., Luo, D.-Y., Gao, Y., and Zuo, D.-C. (2011). Modeling of node energy consumption for wireless sensor networks. *Wireless Sensor Network*, 03(01):18–23.
- Zhu, Z., Olutunde Oyadiji, S., and He, H. (2014). Energy awareness workflow model for wireless sensor nodes. *Wireless Communications and Mobile Computing*, 14(17):1583–1600.