Annotating Executable DSLs with Energy Estimation Formulas

Thibault Béziers La Fosse, Massimo Tisi, Jean-Marie Mottu, Gerson Sunyé

Annotating Executable DSLs with Energy Estimation Formulas

Thibault Béziers la Fosse
IMT Atlantique, ICAM, LS2N
Nantes, France
thibault.beziers-la-fosse@ls2n.fr

Massimo Tisi
IMT Atlantique, LS2N
Nantes, France
massimo.tisi@imt-atlantique.fr

Jean-Marie Mottu
Université de Nantes, IMT Atlantique, LS2N
Nantes, France
jean-marie.mottu@ls2n.fr

Gerson Sunyé
Université de Nantes, LS2N
Nantes, France
gerson.sunye@ls2n.fr

Abstract

Reducing the energy consumption of a complex, especially cyber-physical, system is a cross-cutting concern through the system layers, and typically requires long feedback loops between experts in several engineering disciplines. Having an immediate automatic estimation of the global system consumption at design-time would significantly accelerate this process, but cross-layer tools are missing in several domains.

Executable domain-specific modeling languages (xDSLs) can be used to design several layers of the system under development in an integrated view. By including the behavioral specification for software and physical components of the system, they are an effective source artifact for cross-layer energy estimation.

In this paper we propose EEL, a language for annotating xDSL primitives with energy-related properties, i.e. how their execution would contribute to the energy consumption on a specific runtime platform. Given an xDSL, energy specialists create EEL models of that xDSL for each considered runtime platform. The models are used at design time, to predict the energy consumption of the real systems. This avoids the need of energetic analysis by deployment and measurement on all runtime platforms, that is slow and expensive.

We augment an existing language workbench for xDSLs with an editor for EEL models and a component that computes energy-consumption estimations during model editing. The evaluation shows that EEL can be used to represent estimation models from literature, and provide useful predictions.

CCS Concepts: • **Software and its engineering → Extra-functional properties**; • **Hardware → Power estimation and optimization**.

Keywords: xDSL, Energy Estimation, Cyber-Physical Systems

ACM Reference Format:
Thibault Béziers la Fosse, Massimo Tisi, Jean-Marie Mottu, and Gerson Sunyé. 2020. Annotating Executable DSLs with Energy Estimation Formulas. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3426425.3426930

1 Introduction

Energy consumption has become an important concern in the domain of software engineering during the past decade [44]. Significant efforts aim at reducing the CO_2 emissions of data-centers [52], electricity costs [54, 59], or improving the battery life of smartphones [33, 43]. Energy-aware design of software systems and applications benefits from an estimation of the energy consumption at design time. Using this early feedback, software engineers can perform design choices aimed at energy efficiency, e.g. using the right data structures depending on the context, avoiding energy-consuming code smells [13, 30, 39, 45, 47]. In particular, many tools have been developed for measuring and estimating the energy consumption at the software and middleware levels [7, 15, 34, 40, 50, 55]. Existing approaches from literature address the energy consumption of general-purpose programming languages instructions, e.g. based on LLVM IR [24], Android Bytecode [26], or System call traces [1].

Energy optimization of cyber-physical systems (CPSs) introduces further challenges, since consumption is impacted both by the physical devices and the software running on them, and constrained by limited power supplies. When designing CPSs, software developers need to consider also the physical characteristics of the devices included in the

SLE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8176-5/20/11…$15.00
https://doi.org/10.1145/3426425.3426930

system, since a significant part (usually most) of the consumption goes into software-driven physical devices [48, 51]. If all engineering disciplines have ad hoc tools for estimating energy consumption, tools usable across engineering disciplines are uncommon. As a result, energy-optimization typically requires long feedback loops between experts in several engineering disciplines.

Furthermore, using energy measurement and estimation tools often requires complex software and system tweaking, and competences about energy measurement that the majority of software developers does not have [44, 46].

Finally, providing immediate feedback about energy consumption is more complicated when the application under development is meant to be executed on a large diversity of platforms with their own energetic properties. Indeed, in order to gather energy-related metrics, deploying the application on all platforms, or running several low-level hardware simulators, can be long and expensive. Most existing work targets specific languages, or runtime platforms.

In this work, we argue that models in *executable* domain-specific languages (xDSLs) are an effective artifact for an energy-aware development process for CPSs. Indeed, models are already commonly used during the CPS engineering life cycle [2]. The structure and behavior of executable models are written in a modeling workbench that is typically able to simulate their execution, verify their properties, compile and deploy them on several platforms [4, 16, 21, 35, 56].

We introduce a generic approach for estimating the energy consumption of systems designed by xDSLs. The approach is based on a proposed *Energy Estimation Language* (EEL) for annotating any given xDSL with energy-estimation formulas. An energy specialist writes *Energy Estimation Models* (EEMs), each one defining the energetic properties of the xDSL for *a single specific runtime platform*. The modeling workbench is capable of taking several EEMs into account while simulating an executable model, and predict how much energy it would consume when deployed on its respective platforms. This feedback can help developers identifying energy waste and improving their programs before actually deploying them.

This approach raises the following research questions:

RQ1: Can EEMs associated to xDSLs be used to encode the energy estimation methods in literature?
RQ2: Can the evaluation of EEMs on xDSL execution traces provide accurate energy estimations?

We show the benefits of our approach by a case study, where we define an EEM for an xDSL for Arduino, i.e. ArduinoML. We measure the consumption of Arduino devices using small benchmark ArduinoML models, and we model this consumption in the EEM. We automatically estimate the consumption of three larger ArduinoML application models, obtained by combining these devices. Then we generate code from them and deploy them on the runtime platform to measure and compare the energy consumption to the estimation automatically performed at design time. We detect in our case study an estimation error with an average $4.9\,\%$, between $0.4\,\%$ and $17.1\,\%$.

This paper is organized as follows. Section 2 proposes a running case to exemplify the approach. Section 3 outlines the EEL abstract and concrete syntax, and its semantics. Section 4 experiments on the approach. Section 5 describes the related efforts in literature and finally, Section 6 concludes this paper.

This research work is partially funded by the MEASURE project[1].

2 Running Example

We illustrate the paper with a running example where a developer wants to build a small CPS on top of Arduino.

Arduino is a open-source hardware and software company. It proposes development boards embedding CPUs based on AVR and ARM architectures, but also a cross-platform development environment. A program to be deployed on Arduino boards can be written in any language, as long as it compiles to binary code conforming to the targeted CPU. The standard Arduino development environment supports C and C++. A Arduino program is called a *Sketch*, and consists of two functions *setup* and *loop*. The former is called at the start of the program, and is generally used to initialize the variables, and to define the types of the signal used by the pins of the Arduino board (digital or analogic, input or output). The later defines the behavior of the Arduino board, and is repeated indefinitely.

In this running example, the system that the developer wants to deploy embeds an Arduino board, an infrared sensor and a LED. The behavior to define in the *sketch* is the following: when the sensor detects an obstacle, the LED is turned on for one second and then turned off for another, repeatedly. The CPS has to be produced in several versions, based on different Arduino boards: Arduino Uno, Due, Nano, etc. The developer needs to estimate and possibly improve the energy consumption of her system on all platforms.

If this use-case is intentionally small to be completely addressed in the paper, it shows the cumbersomeness of a standard energy-aware development process. The standard development process requires writing a C program in the Arduino IDE. The main *loop* of the program checks if the signal sent by the sensor is `HIGH`. As soon as it is, a `HIGH` signal is sent to the LED to turn it on, followed by a one second `delay`. Finally a `LOW` signal is sent to the LED to turn it off, and another one second `delay` follows. The written C code may differ among the different Arduino platforms. The developer would manually perform the needed adaptations before deployment. In order to measure the energy consumption, the developer has to deploy each C program on

[1] https://itea3.org/project/measure.html

their related platform, and use specific energy-measurement devices. By reading these measurements, she tries to detect possible inefficiencies and optimize her code.

Instead of developing in C, an developer relying on a xDSL could choose to model the application as an executable model in a modeling workbench such as GEMOC Studio [8], AToM3 [18], Ptolemy [12], or ModHel'X [27], and generate the C code for the different target platforms. This would streamline the development phase but would not yet impact the energy estimation effort to be made for each platform.

For instance, *ArduinoML* is an xDSL for representing structural and behavioral aspects of Arduino systems, embedded in a standalone development environment called Arduino Designer[2]. Figure 1 presents an excerpt of the ArduinoML meta-model. The left part of this meta-model (`Board`) defines the physical properties of the system: which pins are used, by which modules, the level of the signal coming from/going to this pin, etc. The right part of this meta-model (`Sketch`) defines the behavioral properties of the system: what to do with the modules, according to the signals coming from/going to the pins. ArduinoML is executable in GEMOC, since its meta-model defines semantic operations for the language instructions (defined in the `execute()` function). The operations may change the value of runtime properties in ArduinoML, e.g. the `level` of a `Pin`. The GEMOC workbench calls these operations in order to simulate the system execution.

Figure 2 presents the ArduinoML model of the sample program. Its lower part represents the structural aspects of the Arduino system: a LED and an infrared sensor, plugged on the pins 13 and 10 of the board, respectively. The upper part of the model represents the behavioral aspects of the system, previously described.

To optimize it, the developer needs to estimate the energy consumption of this model on the target platforms. The platforms differ in the Arduino boards used (e.g. ProMini, UnoR3), but they may also employ different LEDs or infrared sensors, with their own energy consumption. Typically the developer would generate a different versions of the C code from the model, for each platform, deploy them and execute the system to perform physical measurements. Deploying and performing the measurements with standard power measurement tools can be long, complicated, and expensive, especially if the developer needs to iterate multiple times to tune the performance of her program. The next section introduces our solution to perform this estimation at design time, without deploying the model.

[2]https://github.com/mbats/arduino

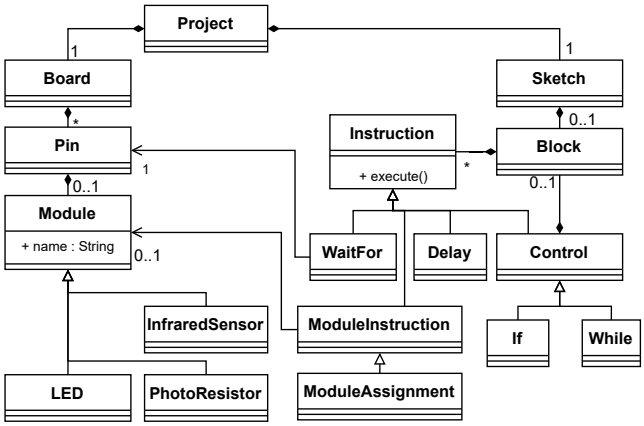


Figure 1. Excerpt of the ArduinoML meta-model

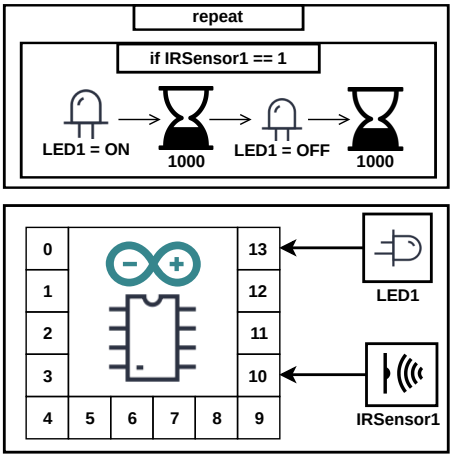


Figure 2. ArduinoML model

3 Energy-Estimation Modeling

3.1 An Energy-Estimation Model

Current modeling workbenches for xDSLs focus on modeling the systems, simulating their execution, verifying their properties, compiling and deploying code on several platforms. They lack features for providing energy estimations at the model level.

We propose EEL, a modeling language to annotate meta-models of executable DSLs with energy-related concepts, allowing the workbench's simulation engine to produce energy estimations. Each model written in EEL is dedicated to one xDSL and one runtime *platform*. We consider a platform as a set of multiple devices, whereas EEL estimates all the executable models, conforming to the annotated meta-model, and to be deployed on any possible assembly of these specific devices. In what follows, we describe an excerpt of the EEM we attach to the ArduinoML xDSL, for estimating its consumption on a specific platform. The platform we put the focus on is based on a Arduino UnoR3 board, using LEDs and infrared sensors with the references L-53MBDL and VMA 330, respectively.

```
1  Platform "ArduinoUnoR3" {
2      // LED L-53MBDL
3      LED.voltage = 5
4      LED.current = 0.00845
5      LED.power = LED.voltage * LED.current
6      // IRSensor Velleman VMA 330
7      InfraredSensor.voltage = 3.3
8      InfraredSensor.current = 0.00235
9      InfraredSensor.power = InfraredSensor.voltage *
           InfraredSensor.current
10     // Board Arduino Uno Rev 3 (ATMega328p CPU)
11     Board.voltage = 5
12     Board.cpuCurrent = 0.0241
13     Board.cpuPower = Board.voltage * Board.cpuCurrent
14     Board.period = 1/16000000
15     Board.nLEDsOn = LED.allInstances()->select(it|it.
           oclContainer().oclAsType(Pin).level=1)->size()
16     Board.nIR = InfraredSensor.allInstances()->size()
17     Board.power = Board.cpuPower + (Board.nLEDsOn *
           LED.power) + (Board.nIR * InfraredSensor.power
           )
18     // Instructions
19     ModuleAssignment.clockCycles = 44
20     ModuleAssignment.duration = ModuleAssignment.
           clockCycles * Board.period
21     Delay.clockCycles = 76
22     Delay.callDuration = Delay.clockCycles * Board.
           period
23     Delay.waitDuration = self.value/1000
24     Delay.duration = Delay.callDuration + Delay.
           waitDuration
25     ModuleAssignment#execute.energy =
           ModuleAssignment.duration * Board.power
26     Delay#execute.energy = Delay.duration * Board.
           power
27 }
```

Listing 1. Excerpt of an EEM for platform made of an Arduino UnoR3, a LED actuator, an infrared sensor

The EEM in Listing 1 attaches energy-related formulas to each meta-class of the ArduinoML meta-model. After specifying the name of the platform, an EEM is a sequence of *estimations*. Each estimation dynamically defines a property in a xDSL metaclass, associating it with a value or an estimation expression.

We first define the voltage, current and power consumption of the LED element and infrared sensor using their technical specifications, as shown at lines 3 to 9. Note that since these values are identical for all LEDs and InfraredSensors in that platform, we can simply refer to them as static properties of the corresponding meta-class. We do not need to estimate other modules in this example but a more complex executable model would require to include all the other sensors and actuators used in the system (motors, photoresistors, etc...).

From lines 11 to 14, we define parameters of the Arduino Board: its voltage, CPU power consumption, and clock period as it impacts the duration of the instructions, and hence their energy consumption. Line 15 counts the number of LEDs turned on using a model query as only those ones will consume energy, and line 16 define the Board total power consumption by summing all the power-consuming devices it holds. Note that we use OCL formulas to navigate the ArduinoML model, that includes the information on system state at runtime.

Then we define the duration of the instructions in the behavioral part of the Arduino language (right part of Figure 1). In this particular example, we want to first estimate the duration out of the number of clock cycles needed to execute the instructions, and the clock period of the CPU. We first define the *ModuleAssignment* duration at line 20, by multiplying the number of clock cycles needed to execute it by the clock period of the Arduino board. For the *Delay* instruction, we also need to consider the delay duration (in seconds) specified by the developer in the ArduinoML model. This duration is queried using OCL, line 23, and divided in order to get milliseconds.

Finally, lines 25 and 26 assign energy-consumption estimations to the `ModuleAssignment` and `Delay` instructions. We estimate the energy consumption of these instructions by multiplying the duration of those instructions by the power consumption of the system. We assign the result of these computations to the `execute` operation of these xDSL instructions.

Given an execution trace of the ArduinoML model, the evaluation component of EEL uses the EEM to compute a global estimation of the energy consumed during that execution. The same execution trace can be used with different EEMs for estimating the consumption of different final platforms before deployment.

3.2 The Energy-Estimation Language

In this section we illustrate the main concepts of our energy estimation DSL, and how it is used for describing the energy consumptions of the targeted xDSLs. An excerpt of the concrete syntax, written with XText [22], is defined in Listing 2, and an excerpt of the abstract syntax is available in Appendix A. We discuss later in Section 3.5 on why using a new dedicated language, instead of OCL and/or Aspect-Oriented techniques to annotate languages with energy-estimation formulas.

The top-level container of this meta-model is the *Platform* element. A single EEM is meant to define the energy consumed by the xDSL on a single platform, defined here. Estimations are defined through the *Estimation* element.

```
1  Platform:
2    'Platform' name=String '{'
3      estimations+=Estimation (',' estimations+=Estimation)
           *
4    '}';
5  Estimation:
6    (post?='post')? target=Target '.'
7      name=(EstimationName | UserEstimationName)
8        ('=' expr=EstimationExpr)?;
9  Target: EClass | EOperation;
10 EstimationName:
11   'duration' | 'frequency' | 'current' | 'voltage' | '
         power' | 'energy' | 'absoluteTime';
12 UserEstimationName: ID;
13 EstimationExpr:
14   (EstimationValue | OCLEstimationExpr |
15   CompositeEstimationExpr);
16 EstimationValue: value=Double;
17 OCLEstimationExpr: query=OCLExpr;
18 CompositeEstimationExpr: TransitionEstimationExpr |
19   TailEstimationExpr;
20 TransitionEstimationExpr: LogisticEstimationExpr | ... ;
21 LogisticEstimationExpr:
22   'logfun('L=[EstimationExpr] ',' k=[EstimationExpr] ',
         ' x0=[EstimationExpr] ',' x=[EstimationExpr] ')';
23 ...
```

Listing 2. Excerpt of the EEL concrete syntax

First, estimations have a *Target*. A *Target* can be either a meta-class of the xDSL or a meta-operation. We decorate meta-classes when we want to declare general energy-related properties on them. For instance, in our example we used an estimation targeting the LED meta-class to define its voltage. An estimation can also target a meta-operation. When an operation conforming to this meta-operation is performed, the estimations targeting it are evaluated, in order to produce an energy consumption estimation.

Estimations have an *EstimationName*. While EEL users can specify their custom *UserEstimationNames*, a set of energy-related estimations (current, voltage, power, energy, frequency, duration) are predefined. These estimations are meant to have special support in the tooling, especially to be used by generic energy-aware visualizations in the modeling workbench. For these estimations, EEL also verifies the consistency of their physical units.

The right-hand side of an estimation is an *EstimationExpression*. Expressions can simply hold an *EstimationValue*, defined by a Double, or be more complex.

OCLEstimations contain an OCL query. This query is evaluated in the context of the targeted element, and the value is assigned to the estimation. While we currently use standard OCL (from the OCL Eclipse project), the connection with OCL is completely modular, and the language is suitable for integration with different flavors of OCL. For instance uncertainty-aware OCL [36] may be used to specify estimations that are better represented by probability distributions.

Besides inheriting the expressive power of OCL, the language supports domain-specific composition of estimations by extending the *CompositeEstimation* meta-class. The extension, performed in Java, enables the integration of existing mathematical libraries for representing complex functions, calculus, or numerical estimations. For instance we can use it to represent several sigmoid functions from literature to model the transition from one state to another (like *LogisticEstimationExpr* in Listing 2). We can also provide several decreasing tail functions, typically used in energy estimation to represent phenomenons like tail energy, i.e. hardware-induced energy consumption corresponding to an activity happening after a device is used (see Section 4.1).

Finally, a special estimation *absoluteTime* is used to refer to the clock value, stored in the execution trace. This value is useful to calculate durations of events during the execution, e.g. the delay of user input. We will discuss these issues in detail in Section 3.5.

3.3 Evaluation Semantics

The entry point of the semantics for EEL are the energy estimations attached to meta-operations. The expressions associated to these estimations are evaluated every time the semantic operation is called, *by default immediately before*. Since the state of the model before performing the operation is known, the changes that the called operation will apply can be anticipated. And thus the energy consumption of the DSL operation can be estimated.

Each estimation is lazily evaluated when needed. Estimations are built on top of each other, in a hierarchical fashion (reference cycles between estimations are not allowed). Hence, an estimation can be represented as a tree, whereas the first estimation is the root, and depends on the values of its children. When evaluating an estimation, the tree is traversed depth-first, in post-order. Figure 3 shows the estimation for a `ModuleAssignment.execute()` operation, in a tree shape, based on Listing 1. Thus, to estimate this instruction, the leaves are first evaluated. They are defined as simple double values, and used later by the composite estimations of the trees. As an example, `IR.power` computes the product between its two leaves `IR.current` and `IR.voltage`, for an estimated power consumption of $0,00775$ Watts for the Infrared Sensor. The same reasoning applies until reaching the root of the tree. This represents an estimated energy consumption of 4.68×10^{-7} Joules for the `ModuleAssignment.execute()` instruction, considering that there are only one LED and Infrared sensor in the system.

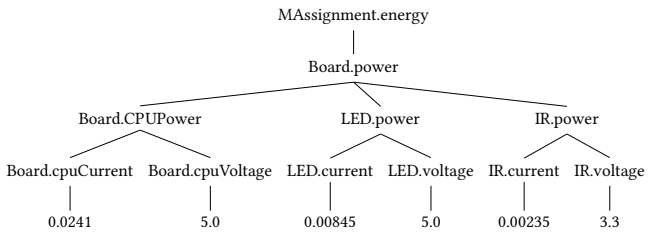


Figure 3. Evaluation tree for the `ModuleAssignment.execute()` operation

As some durations cannot be anticipated, e.g. because of user inputs in the trace, it is sometimes convenient to perform estimation at the end of the execution of a semantic operation. EEL provides a specific keyword for the purpose, `post`. Any estimation marked by `post` is executed right after the conclusion of a semantic operation. Note that if an estimation depends on a `post` estimation, then it will also be calculated after the execution of the semantic operation.

As an example, considering the `WaitFor` instruction defined in ArduinoML. This instruction puts the board on sleep mode until a pin's signal changes. Computing the energy estimation of this instruction requires to measure the duration of this wait and then to use it along with the power drawn during it, to compute an energy. This can be done by using the `Board.absoluteTime` estimation. This value has to be stored in an estimation property before performing ArduinoML's `WaitFor` instruction. Then, is has to be evaluated a second time at the end of the `WaitFor` instruction, using the `post` keyword of EEL. The difference between the values of these two Estimations is the duration of the `WaitFor` instruction, which is then used to estimate the energy it consumed.

Furthermore, considering the EEM in Listing 1, computing the power drawn by LEDs is performed, line 17, by multiplying the number of LEDs turned on by the individual power of a LED. In this specific example, this works since a single type of LED is used for this platform. However, if different types of LEDs are used in the system, the individual power of each LED may change. To estimate such platform, each LED has to be properly identified in the Arduino model (e.g., by its name), so that these identifiers can be used in the EEM to provide per-LED power consumptions. Listing 3 shows an example of such EEM. Two types of LEDs are defined. OCL is used to compute the number of LEDs of each type currently turned on, and the total power consumed by LEDs of all types can be computed. Nevertheless, it implies that the developer using ArduinoML and the energy specialist designing EEMs conform with the same naming conventions. In fact, if the LED's type is not properly defined in the ArduinoML model, then EEL has no way of knowing which type it is.

3.4 The Energy-Estimation Modeling Process

EEL is meant to be used in the process illustrated by example in Figure 4. The developer designs (1) the Arduino application model to be deployed on several platforms (2).

We introduce a new actor called *Energy Estimation Specialist*. This actor knows the xDSL (ArduinoML in our case) and the platforms on which the models can be deployed. The specialist provides the developer with one *Energy-Estimation Model* (EEM) for each platform (3). Each EEM defines the energy consumption of the ArduinoML operations for its related platform.

To estimate the energy consumption, the developer needs a set of execution traces of her ArduinoML model, storing the timed execution of the semantic operations of the xDSL (e.g. the *execute* operation) and the state of variables at these times. The production of execution traces for an executable DSL is outside the scope of this paper. They can be derived e.g., by executing the simulator on a set of benchmarks (4), by different trace synthesis methods, or by reusing/adapting real-world traces for previous executions.

An execution trace can finally be analyzed by the EEM evaluation component. This estimates (5) the energy each Arduino platforms (based on different Arduino Device, e.g. ProMini or UnoR3, and Modules, e.g. LED L-53MBDL or L-7113ID) would consume when running the Arduino program.

```
1  Platform "DifferentLEDs" {
2      // LED L-53MBDL
3      LED.voltage = 5
4      LED.53MBDL_current = 0.00845
5      LED.53MBDL_power = LED.voltage * LED.53
           MBDL_current
6
7      // LED L-7113ID
8      LED.7113ID_current = 0.00765
9      LED.7113ID_power = LED.voltage * LED.7113
           ID_current
10
11     Board.numberOf7113ID = LED.allInstances() -> select (
           it | it.oclContainer().oclAsType(Pin).level = 1) ->
           select ( it | it.name.substring(1,6) = '7113ID') ->
           size()
12     Board.numberOf53MBDL = LED.allInstances() -> select
           (it | it.oclContainer().oclAsType(Pin).level = 1) ->
           select ( it | it.name.substring(1,6) = '53MBDL') ->
           size()
13     Board.powerOfLEDs = Board.numberOf53MBDL * LED
           .53MBDL_power + Board.numberOf7113ID * LED
           .7113ID_power
14 }
```

Listing 3. Excerpt of an EEM for a platform with different types of LEDs

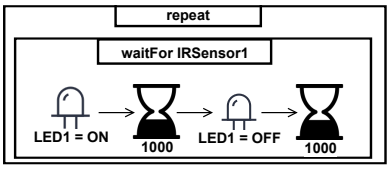


Figure 5. Updated Behavioral part of the ArduinoML Model

An immediate feedback is given to the developer about the energy that would be consumed on the final platforms. It can hence help her doing the best design choices for energy efficiency (6).

Applying the process to our example in Figure 2, the developer may analyze the trace of a benchmark simulation that sets to 10 seconds the user time before triggering the infrared sensor. The EEM evaluation component exploits the EEM in Listing 1 to estimate for this trace an energy consumption of 1.6044 J on the platform featuring an Arduino UnoR3.

The ArduinoML developer can immediately detect an elevated consumption, and try to improve the behavior by replacing the `if` ArduinoML instruction by a `waitFor` instruction as shown in Figure 5. Instead of actively checking this sensor, the Arduino is put to sleep until the sensor detects a change.

The same benchmark is simulated again, and the evaluation component analyzes the new trace, estimating an energy consumption of 0.924 11 J. That validates the developer choice without requiring her to deploy the new ArduinoML model, while keeping the same EEM.

3.5 Discussion and Limitations

EEM definition. Defining an EEM is not trivial. The energy-estimation specialist needs specific measurement tooling, knowledge about the platform on which the executable models will be deployed, and knowledge about the xDSL. Several parameters can be retrieved from the technical specifications of the devices used in the system, usually provided by manufacturers. Note that, when the same devices are used in another platform, their description in EEL can often be reused (e.g., in the case of Figure 4 with two platforms with the same IRSensor VMA 330).

A typical way of estimating the EEM is by designing multiple small xDSL models, to study the consumption curves of a single language element in its possible uses. Assisting the energy specialist in writing EEM models by producing the right executable models and analyzing the measurements is the subject of our future work.

EEL and aspect-oriented programming. EEL is meant to attach energy-related concepts to the classes and operations of executable languages. These concepts are then evaluated dynamically (at runtime, or on execution traces), and do not impact the behavior of the executable model estimated. Furthermore, specific keywords available in EEL can specify whether an estimation should be computed before, or after the targeted operation. This approach might look close to what aspect-oriented programming (AOP) and software instrumentation propose, and thus might hamper the will of learning a new DSL, especially since robust and mature frameworks are available. EEL directly offers vocabulary and mathematical functions for specifying concepts from the domain of energy estimation, and make OCL expressions directly applicable for that purpose. We believe that it is easier for energy-specialists, eventually oblivious to AOP, to write completely declarative specifications of energy-estimation formulas, instead of using an imperative programming language such as Kermeta [29].

Non-determinism. An EEL model estimates the consumption of each given execution trace of the system. If the system

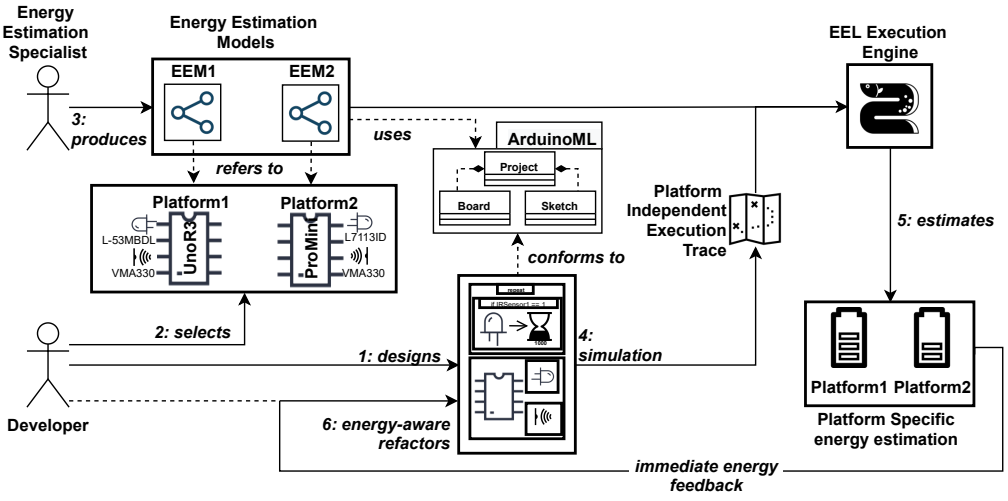


Figure 4. Process for the estimation of Arduino energy consumption at design time

has some non-deterministic behavior, or requires user interaction, the results of non-deterministic choices, user input values and timings will be stored in the timed sequence of events of the trace. Thus, since EEL estimates these deterministic traces then it is not impacted by non determinism in the simulator.

Furthermore, if the execution of a same model on both the simulator and the final platform differ, e.g. because of non determinism, then the estimation provided by EEL, based on the simulation, might not be accurate. Using a simulator that reflects the exact behavior of the final platform would solve this issue.

Another source of non-determinism is concurrent execution of semantic operations, that typically impacts the relative order of events. Again timings and effects of concurrency are stored in the execution trace, that can be deterministically estimated by EEL.

Quality of traces. The quality of the right execution traces (e.g. from the right benchmarks) has a strong impact on energy estimation. While the production of the right execution traces and benchmarks lies outside the scope of this paper, we report here some observations.

If traces are derived by a simulator, the simulator should reflect as much as possible the behavior of the final platform. Non-deterministic behavior and effects can differ between the final platform and the simulator, because of a different implementation, or because of variations in the execution environment on which the models are executed.

To be usable by EEL, execution traces should contain the entities executed, and also store execution times. As durations can be different in the simulator than in the final platform, if traces are derived from a simulator, then runtime system times should be estimated. Time estimation is possible on EEL, by defining a `duration` estimation. The estimation should access the `absoluteTime` element to retrieve the execution-trace time, and apply a composite or OCL estimation expression to perform the conversion.

Probabilistic EEMs. Current estimations by EEL return a numeric value for each estimation. By using uncertainty in OCL from [36] this value can be associated with uncertainty measures. In case of non-deterministic behavior, a larger number of traces can be energy-estimated to derive a probability distribution.

An alternative approach would be encoding probabilities directly within the EEM model. The resulting probabilistic EEM would compute and store estimations as complex objects representing a full probability distribution. This is especially feasible when the energy-estimation specialist can reasonably estimate the probability of the non-deterministic choices or user interactions. This is a possible line for future work.

3.6 Implementation Details

We implemented the editor and evaluation component of EEL as an extension to GEMOC Studio[3]. GEMOC Studio is a language and modeling workbench for model design and execution [5, 9, 10]. Its execution engine embeds a generic trace constructor, an omniscient debugger, and several extension mechanisms. The ArduinoML integration within GEMOC enables the execution of Arduino models in a simulator, as well as code generation. For our experimentation, we directly extend the GEMOC trace generator to estimate the energy consumption during the simulation without waiting the full trace to be completely produced.

Our implementation relies on the *Java Engine* of GEMOC Studio [8]. This engine is dedicated to operational semantics directly written with Java, Xtend [20], or Kermeta [29]. The executable semantics is a sequence of calls to the semantic operations of the xDSL (e.g. `execute()` in ArduinoML). It is composed of several operations called during the execution of models, including the followings: (1) `initialize` is called before the execution, and performs the loading of the model to be executed; (2) `beforeStep` is called before executing operations annotated with `@Step` in the operational semantics; (3) `afterStep` is called after executing operations annotated with `@Step` in the operational semantics.

We extend GEMOC's Java Engine with an addon that performs additional behavior on top of the existing operations. The *initialize* extension simply consists in loading the energy estimation model provided by the *energy estimation specialist*, thus making it available during the xDSL model execution. Estimations are performed in the *beforeStep* and *afterStep* depending on the `post` keyword.

4 Evaluation

In this section we evaluate our approach against the research questions that motivated this paper. The first part of this evaluation shows how EEL can model existing domain-specific energy estimation approaches from literature, answering **RQ1**. The second part presents a workflow where we use EEL to estimate the energy consumption of Arduino systems, answering **RQ2**.

4.1 Expressiveness

In this section we evaluate the expressiveness of EEL, i.e. its ability to model energy-estimation approaches existing in the literature. Since energy-estimation profiles for xDSLs are not currently available or not using Model-Driven Engineering [49], we select from literature well-known *per-instruction energy-estimation profiles* for general-purpose languages. EEL can represent those energy estimation-profiles and evaluate them, when it is given execution traces of their respective systems. While we can not argue about the complexity of future energy-estimation profiles for xDSLs, this

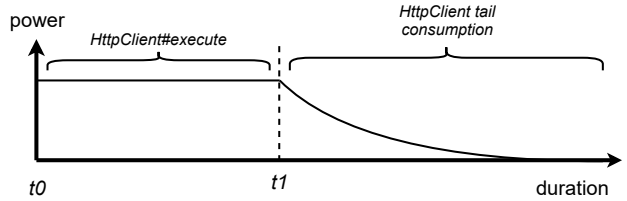
[3] https://github.com/atlanmod/eel

shows that EEL is expressive enough to represent current ones.

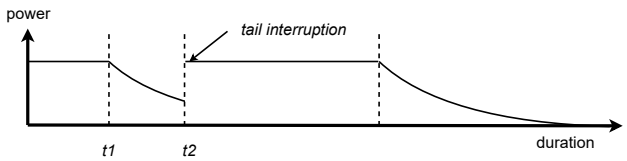
Shuai Hao et al. propose Software Energy Estimation Profiles (SEEPs). SEEPs associate Android instructions with Hardware energy costs [26]. In previous work, they show how to calculate per-line energy consumption [32], and introduce *tail energy consumption*. Tail energy is a hardware-induced energy consumption, corresponding to an activity happening after a device is used. Such energy is calculated using a mathematical function provided by the hardware manufacturer. They illustrate this with an excerpt of Android code featuring the `HttpClient.execute()` instruction. Figure 6a shows an example of tail energy consumption occurring after using `HttpClient.execute()`.

Estimating the energy consumption of this instruction as well as the tail energy consumption do not represent a challenge. However, the EEL model has to consider eventual interruptions of the tail consumption. The Expressiveness of EEL allows us to model such event. For instance, Listing 4 shows an equivalent SEEP modeled with EEL to estimate the energy consumption of this Android instruction, and considering the tail energy consumption with interruptions. We detail this EEM.

Line 1 captures the absolute time when the instruction `HttpClient.execute()` is called, using the *absoluteTime* global property. Line 2 computes the energy consumed during the execution of this instruction, which corresponds to the area under the power plot in Figure 6a between *t0* and *t1*. We simplify this computation here to only focus on the tail. Line 3 computes the tail energy consumption. As stated before, this energy consumption is defined by a mathematical function provided by the manufacturer of the concerned device. We define it as *tailFunction(duration)*, a fictional mathematical function taking the duration of the tail as a parameter, but a more complex function, eventually available in EEL (logistic, exponential, integral, etc.), could have been used. This duration is computed by subtracting the absoluteTime when the `HttpClient.execute()` instruction last ended, corresponding to *t1* in Figure 6b, to the absoluteTime at which it started last, *t2* in Figure 6b. This corresponds to the interrupted tail's duration, and can be used to estimates the tail's consumption. Then, line 4 sums the tail energy to the call energy, to compute the energy of the instruction, and finally line 5 updates the time at which this instruction ended. Since these operations are performed sequentially, the time elapsed since the last usage of the wi-fi card is updated *after* computing this tail energy consumption. Such EEM could be attached to an executable language defining Android Java classes (using MoDisco [11], for instance).



(a) Tail energy consumption



(b) Tail interrupted

Figure 6. Tail consumption behaviors

Neville Gretch et al. associate LLVM IR instructions with constant energy costs for a given platform, and statically estimate the energy consumption of programs relying on control flow graphs [24]. For such low-level consumption models, EEL can simply list the energy consumptions of each LLVM IR instructions as shown in Listing 5. The EEL evaluation component will sum the consumptions of all instructions in the trace.

The approach presented by Karan Aggarwal et al. analyzes system call traces for estimating the impact of software changes on *power* consumption [1]. They build linear regression models, that associate a power usage to each system call instruction. They model the global power usage as the average power of all system calls, weighted by their number of appearances in the execution trace. Their approach can be used to estimate the energy consumption of an application with EEL, as described in Listing 6. Each system call is attached to an average power consumption. EEL counts the times systems calls are executed, and it derives the average power consumption. The total energy consumption is computed multiplying the average power and the elapsed time in the system.

4.2 Estimation Accuracy

In order to answer to **RQ2**, we evaluate our approach in predicting the energy consumption of Arduino systems. First, we describe the runtime platforms where we will deploy the ArduinoML application models, and we describe how an *energy estimation specialist* would define EEL models for those platforms. Then we consider a real-world ArduinoML model. We estimate, using EEL, the energy consumption of this ArduinoML model when deployed on those platforms. To check the accuracy of the energy estimation, we measure and compare the actual consumption of that system by generating C code, deploying it and performing hardware measurement. Finally, we follow the same process for the two sample systems in Figures 2 and 5.

4.2.1 Deployment Platforms. We consider two deployment platforms. A first platform is based on an Arduino

UnoR3 board. The platform defines a specific device for every module in the ArduinoML model. All LED entities declared in the structural part of the ArduinoML application models (such as the one of Figure 2 or Figure 8) having the color blue are deployed as Kingbright LEDs with the reference L-53MDBL. Infrared sensors are deployed as Velleman Obstacle Avoidance sensors with the reference VMA 330, servo motors as TowerPro SG90 and photoresistors as GL55. We will refer to this platform simply as *UnoR3*.

A second deployment platform has two differences w.r.t. the first one: instead of featuring a Arduino Uno R3 board, it uses a Arduino Pro Mini, and instead of using a blue LEDs, it relies on Kingbright L-7113ID red LEDs. We will refer to this platform simply as *ProMini*.

```
1  HttpClient#execute.absoluteTimeLastCallStart = HttpClient.
       absoluteTime
2  HttpClient#execute.callEnergy = HttpClient.cpuEnergyCost
       + HttpClient.wifiEnergyCost
3  HttpClient#execute.tailEnergy = tailFunction(HttpClient#
       execute.absoluteTimeLastCallStart − HttpClient#
       execute.absoluteTimeLastCallEnd)
4  HttpClient#execute.energy = HttpClient#execute.
       callEnergy + HttpClient#execute.tailEnergy
5  HttpClient#execute.absoluteTimeLastCallEnd = HttpClient.
       absoluteTimeLastCallStart + HttpClient#execute.
       duration
```

Listing (4) EEM for Android HttpClient

```
1  call#execute.energy = ...
2  op#execute.energy = ...
3  memload#execute.energy = ...
4  ...
```

Listing (5) EEM example for LLVM IR

```
1  mmap2.power = ...
2  mnap2#execute.count = mnap2#execute.count + 1
3  open.power = ...
4  open#execute.count = open#execute.count + 1
5  ...
6  app#execute.startTime = app.absoluteTime
7  post app#execute.calls = mnap2#execute.count + open#
       execute.count + ...
8  post app#execute.power = ((mmap2.power * mnap2#
       execute.count) +
9    (open.power * open#execute.count) + ... ) / app#execute.
         calls
10 post app#execute.energy = (app.absoluteTime − app#
       execute.startTime)
11   * app#execute.power
```

Listing (6) EEL Power estimation model example for system calls

4.2.2 ArduinoML EEM.

Considering the couple platform/xDSL, the energy specialist will consider one by one the meta-classes of ArduinoML meta-model to describe with EEL what would be their impact on the global energy consumption.

The energy specialist may use benchmarks to estimate the power curves on the platform. Benchmarks are typically made of several small ArduinoML models. Each ArduinoML model focuses on a single module, in order to understand how its presence impacts the energy consumption of the entire Arduino system. For more complex platforms, benchmarks focusing on the interaction between pairs of components are needed, to estimate if it can have effects of the energy curves.

We provide in EEL estimations of the energy consumption of the following meta-classes: Board (in two states: running, and sleeping), LED, InfraredSensor, ServoMotor, PhotoResistor. We individually deploy a specific ArduinoML model and perform measurements for each one of these meta-classes, in order to produce energy-consumption curves that we model with EEL. The curves used to define the EEM are available in Appendix B. In the simplest cases, producing an energy estimation formula can be done by simply averaging the power consumed by the meta-class measured. For instance, the Arduino UnoR3 board in running state shows an average power of $122.5\,\mathrm{mW}$ (Appendix B.a), thus we model this power consumption with EEL, and attach it to the `Board` meta-class of ArduinoML.

To measure the consumption of a Module (e.g., a LED), we need to compare the consumption of the platform with and without this Module activity (*cf.* Appendix B.a and Appendix B.b). Some meta-classes require more reasoning in order to be properly estimated and modeled with EEL, e.g. the photoresistor and the servo motor . In fact, the power consumption of the photoresistor is not constant, but depends on the intensity of the light it measures (*cf.* Appendix B.c). The technical specifications of the photoresistor provided by the manufacturer define the resistance of the module as a linear function of the light it measures. We model this specification in OCL as a linear function of the signal received from the analog pin.

The energy consumption of the servo motor requires a more complex formula: the energy consumption peaks during the first degrees of the rotation, falls, and finally remains (approximately) constant until the rotation finishes (*cf.* Appendix B.f). We use three mathematical functions to estimate this energy consumption. A first linear function estimates the peak of power at the beginning of the rotation, and is applied to the first degrees of the rotation. A second exponential function estimates the power dropping. Finally a constant power function estimates the power until the rotation finishes. The duration of each of those steps is calculated using the technical specifications of the servo motor used. These functions are also defined using OCL.

All these estimations are modeled with EEL to produce the EEM of the first platform, partially shown in Listing 1. Finally we replicate the process with the second platform. We produce a different EEM for estimating ArduinoML on the new platform. The energy-consumption curves are very similar, but the new platform impacts the parameters in the EEL model for the Board and LED elements. In fact, in this second EEL model the Board and LED power consumptions are respectively 32.5 mW and 11.5 mW smaller than in the first EEL model.

4.2.3 ArduinoML Model Estimation. We use the EEM model just defined to estimate the energy consumption of the two sample ArduinoML models (Figure 2 and Figure 5, respectively labeled *IfTempo* and *WaitForTempo*) and of a third ArduinoML model (Figure 8), from a realistic application. The structural part of this ArduinoML model relies on four modules: a button, a LED, an infrared sensor and a servo motor. This model represents an automatic door, defining the following behavior: once a button is pressed, the motor starts rotating. This rotation is divided in thirty 6° rotations (180° total) separated by 90 ms delays[4]. If at any time the infrared sensor detects an obstacle, the rotation is interrupted, and a LED blinks four times, during 500ms, as a warning. The user can then resume the rotation by pressing the button.

We simulate the execution of this model within GEMOC studio and use our EEL evaluation component with, as an input, the EEMs. In the benchmark we manually simulate the button press and the obstacle presence. This execution produces energy estimations for each operations of the ArduinoML language.

[4]While the delay is not perceived by the user of the door, separating the rotation in small steps is customary for reducing the speed of fixed-speed servo motors

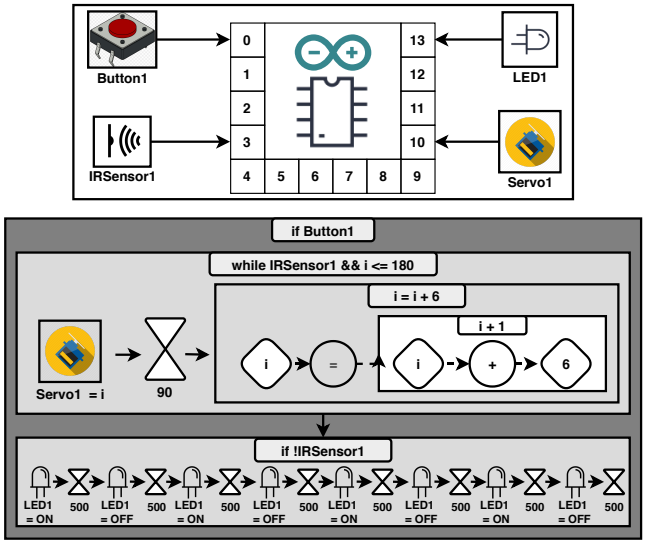


Figure 8. ArduinoML model of an automatic door

Table 1 shows the results of the estimations. The energy estimation for the UnoR3 platform is decomposed into the first four rows. The first row corresponds to the energy estimation of the rotation of the servo motors. Second and third rows correspond to the blinking behavior, estimating the energy consumed when the LED is respectively off, and on. The fourth row shows the totals. The columns show duration of the benchmarks, number of measurements, measured energy, estimation and accuracy.

The following two lines show the result of the estimation of the *IfTempo* and *WaitForTempo (WFT)* applications on UnoR3. The rest of the table shows all the previous estimations, this time applied to the second platform, ProMini. Note that for each case study, estimations for both platforms are computed and shown to the developer at the same time.

4.2.4 Deployment and Physical Measurements. In order to validate these energy estimations, we proceed to physical measurements. We first generate the C code of the three ArduinoML models using Acceleo [38] model-to-text transformations. Structural information of the ArduinoML model is used to configure the `setup()` function in the Arduino code, whereas the behavioral information of the ArduinoML model is used to write the `loop()` Arduino function. The generated code is then deployed via USB through the Arduino IDE. The Measurement column of Table 1 decomposes the energetic measurements made on the platform deployed for the door system and for the two other systems. The last column computes the accuracy of the estimation w.r.t. the physical measurements.

4.2.5 Discussion. As shown in Table 1, total energy estimations are consistently closer than 10 % to the ground truth, thus answering positively to **RQ2**. In some cases we obtain very high accuracy, especially on the ProMini platform (99.0% for IfTempo and 98.9% for WaitForTempo). For the larger use case the two platforms are estimated with

Table 1. Comparison of the measures and estimations for the Arduino models

	Platf.	Dur.	#Meas.	Meas.	Estim.	Accur.
Rotations	UnoR3	2.7 s	507	0.77 J	0.67 J	86.7%
LED off	UnoR3	2.0 s	378	0.32 J	0.32 J	98.5%
LED on	UnoR3	2.0 s	376	0.41 J	0.40 J	99.6%
Total Door	UnoR3	6.7 s	1261	1.51 J	1.40 J	92.7%
IfTempo	UnoR3	12.98 s	986	1.72 J	1.60 J	92.9%
WFT	UnoR3	12.93 s	982	1.02 J	0.92 J	89.8%
Total	UnoR3	32.61s	3229	4.27 J	3.93 J	92.1%
Rotations	ProMini	2.7 s	487	0.70 J	0.58 J	82.9%
LED off	ProMini	2.0 s	377	0.25 J	0.25 J	97.6%
LED on	ProMini	2.0 s	376	0.31 J	0.32 J	97.0%
Total Door	ProMini	6.7 s	1240	1.23 J	1.16 J	94.0%
IfTempo	ProMini	13.00 s	992	1.49 J	1.50 J	99.0%
WFT	ProMini	12.99 s	990	0.72 J	0.73 J	98.9%
Total	ProMini	32.69 s	3222	3.47 J	3.38 J	97.4%

similar accuracy (92.7% for UnoR3, 94.0% for ProMini). If our estimations are accurate, this is also due to the high quality of the simulator in which we performed this evaluation. ArduinoML's operational semantics defined in GEMOC Studio are fine grained, and thus can be accurately estimated.

Performing accurate measurements of the Arduino final platform requires (1) a stable power supply, and (2) an accurate current sensor. In order to have a stable power supply, we power the final platform with the deployed ArduinoML model through the $5\,\mathrm{V}$ port, which is wired to a second Arduino that outputs a regulated $5\,\mathrm{V}$. This second Arduino measures the current it delivers using a INA219 current sensor. This sensor has a high accuracy ($0.5\,\%$), and can send data over the Arduino's I^2C interface, which can then be easily gathered and analyzed by a computer through the Arduino's Serial port. We perform measurement every $3\,\mathrm{ms}$ for this evaluation.

If the estimation is very close to the energy measured when the LED blinks (resp. $98.5\,\%$ and $99.6\,\%$), it is slightly less accurate for the motor's rotation ($86.5\,\%$). This is due to several factors: (1) the energy consumption behavior of the motor is different when performing small and large rotations (2) the frequency at which measurements are performed is not sufficient, and power spikes happen between measurements (3) the behavior of the servo motor is non deterministic, and our EEM cannot estimate it accurately.

5 Related Work

In this section we present the main related work on energy estimation, model-driven approaches for energy efficiency, and properties for executable models.

Instruction-level Energy Estimation. Many tools have been developed and are currently used for assessing of the energy consumption of software and applications [17, 23, 40, 49]. While some of them rely on per-instruction energy estimation, none of these approaches are focused on DSLs. These approaches were mentioned in Section 4, and are detailed here: Shuai Hao et al. propose an approach for providing energy estimations to developer in order to help them implementing efficient applications [26]. An instrumentation is first performed in order to determine which paths of the code are traversed. Then, using a per-instruction energy profile, the energy consumption of the program is estimated. The approach presented in this paper is accurate (within 10% of the ground truth), and shows that it can be used for improving energy efficiency. Neville Gretch et al. statically estimate the energy consumption of compiled programs, in order to provide immediate feedbacks to developers without needing hardware or software measuring tools [24]. They attach energy consumption information to instructions compiled to LLVM IR, and rely on *cost relations* for representing the cost of running the program considering its inputs. Using their approach they can estimate the consumption within 10% and 20% of the ground truth for the two platforms they measured. Karan Aggarwal et al. present an approach based on system call traces for predicting the impacts of software changes on energy consumption [1]. Power consumption is measured during test execution, and associated to system call traces. A linear regression model is built in order to associate a subset of system calls with power consumptions. This model is then used to estimate the consumption of the application based on its system calls. The accuracy of the estimations vary from 68.07% to 87.7%. This approach is interesting here, as it uses execution traces for performing energy estimations, and shows that it can be reliable.

These papers strongly influenced our line of research. We aim at generalize and model instruction-level energy estimations, especially applying them to domain-specific instructions. We overpass their approach by proposing a generic language EEL to extend any xDSL when their models for energy estimation are language specific.

Model-Driven approaches for Energy-Efficiency. Little work has been done, to our knowledge, in the area of Model-Driven Engineering for energy efficiency. We describe existing approaches relying on models and DSLs for energy optimization of particular software systems.

In a previous work we proposed an approach to attach OCL expressions to the meta-elements of xDSLs [31]. These OCL expressions are mathematical formulas defining energy estimations. When models conforming to the xDSL are executed, the OCL expressions are evaluated in order to calculate energy estimations. We extend this approach and propose an external language with an ad-hoc semantics for writing complex estimations, especially when they need to use time and previous states of the system for computations. It is applicable to systems with non-determinism, user inputs and concurrency.

The other model-driven approaches we are aware of are specific of a certain domain, and/or cannot be applied to different languages. Brian Dougherty et al. propose an approach for modeling Cloud Infrastructures, transforming them to CSP and optimizing with a constraint solver for reducing the energy consumption [19]. Thomas Kurpick et al. use models of cyber-physical systems for designing energy-efficient buildings. The building models are constrained with OCL rules, from which analysis algorithms are derived and executed at runtime. Chris Thompson et al. use DSLs for modeling mobile applications, the code generated is instrumented for enabling power measurement, and executed in emulators to estimate the energy consumption [57]. This approach performs energy estimations at design time using simulation. Luca Berardinelli et al. propose an extension of the Agilla Modeling Language (AML) instruction set with an instruction for retrieving the battery voltage of CPS Sensors at runtime [6]. The battery data measured is then used for improving the design of the CPS, and predicting its

energy consumption using fUML executable models. However an energy model is built using real-time data, and is later used for predicting energy consumption of a system, through a simulation defined with fUML. This approach differs from ours, as it requires the usage of a specific language extension, the energy-related metrics only consider the hardware, and offer little composability for complex estimations. Chiraz Trabelsi et al. propose a meta-model for power estimation, that they use in the context of embedded system design [3, 58]. They simulate the execution of systems-on-chip, and according to the activity occurring on the components, they estimate a power consumption of the system. Their approach fosters power estimation through simulation at design time, in order to ease the development of systems. However, it is domain specific as it only considers systems-on-chip. Marcio Oliveira et al. perform design space exploration on UML representations, in order to optimize Java software, and multi-processor system-on-chip [41, 42]. They estimate the resource consumptions of their models, in order to choose the best modeling solution. Their approach is static, whereas ours is dynamic, and considers execution traces.

None of those approaches attaches estimations of energy consumption at the language level in order to estimate any xDSL. The list shows however that MDE can be a useful asset for energy efficiency, either through design space exploration, simulation, code generation, model transformation, or models at runtime.

Properties of executable models. Several alternative methods have been considered for representing the computation of properties of executable models, such energy consumption. Eric Cariou et al. propose an approach for weaving business code into executable models [14]. Business operations are associated with executable elements, and executed before, during, or after the execution of the targeted element. David Mosteller et al. present a simulation environment for prototyping and running DSMLs, providing a simulation feedback through a graphical view [37]. Ábel Hegedüs et al. present a generic replay mechanism for execution traces of dynamic modeling languages [28]. This proposals are suitable candidates for analyzing and estimating energy consumption of xDSMLs models. We base our approach and implementation on the executable semantics and extension points on GEMOC, in order to integrate our tool with the GEMOC ecosystem. Gayane Sedrakyan et al. propose an approach for enriching executable models with user feedback [53]. Their approach gives the user visual information about the models they are executing, hence easing the design and understanding of their models. It requires important feedback from users which are not energy specialist whereas our approach separates the developer and energy specialist roles to benefit independently of their own expertise. The Object Management Group (OMG) introduce the Structured Metrics Meta-model (SMM) [25]. SMM enables the representation of properties, measurements, and entities performing measurements. Entities can be composed in many ways, hence fostering reusability in all engineering domains. It is a perspective to use SMM to persist the energy in trace but it does not provide help to design energy-consumption models.

6 Conclusion and Future Work

In this paper we presented an approach for estimating the energy consumption of executable models when deployed on their target platforms. We introduced EEL, a language enabling the specification of the energy-related properties of a system. Each EEL model attaches energy-related concepts to the meta-elements of an xDSL for one specific deployment platform. Execution traces of models conforming to the xDSL can be used along with an EEL model, to estimate the energy consumption of this executable model, on its platform. We propose a concrete syntax for writing EEL models and evaluate our approach by estimating several ArduinoML models. The results show that Arduino estimations written with EEL are between $0.4\,\%$ and $17.1\,\%$ of the ground truth, and $4.9\,\%$ on average. Using this immediate feedback, the developer can improve the energy efficiency of its models before deploying them. These predictions do not require any knowledge about energy consumption or measurement for the developer and require little effort to be produced.

EEL is meant to be an interface between the estimators of energy-consumption functions and system developers. While in this paper we focused on the syntax, semantics, and integration of the language with the modeling workbench, in future work we intend to focus on automatically producing EEL models from a set of measures of the system consumption curves. Furthermore, we want to be able to reuse EEL models more easily, first across platforms. Reusing EEL models is currently done using copy & paste, but future implementations will enable the definition of *libraries*. Any EEL model could be imported into an other one as a library, and specific concepts in this model could be selected. As an example, if two platforms embed different modules but share the same LED definition, the second platform should be able to import the LED definition available in the first EEM model, using a line of EEL code similar to `"from EEM_platform1 import LED"`. Reusing EEL models across languages is also considered. Making EEL meta-model agnostic is challenging, but can be useful to estimate platforms on which models defined with different languages will be deployed. Moreover, we want to improve the visual feedback within the modeling workbench, to automatically highlight the parts of the model that are the main culprits of energy waste. Also, EEL could be integrated in other environments than GEMOC Studio. In fact, environment used by CPS engineers such as Simulink or Capella could benefit from EEL.

A EEL Abstract Syntax

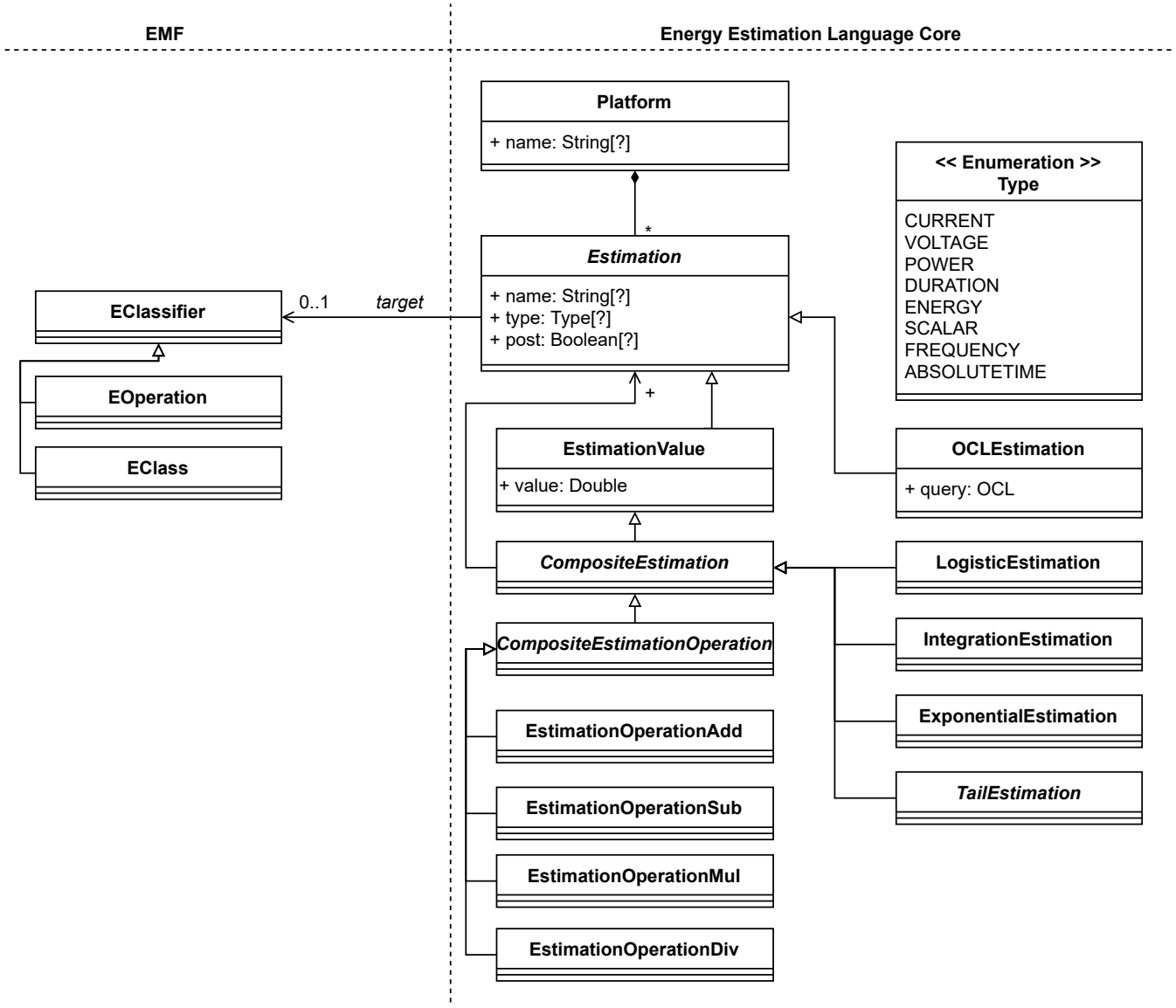
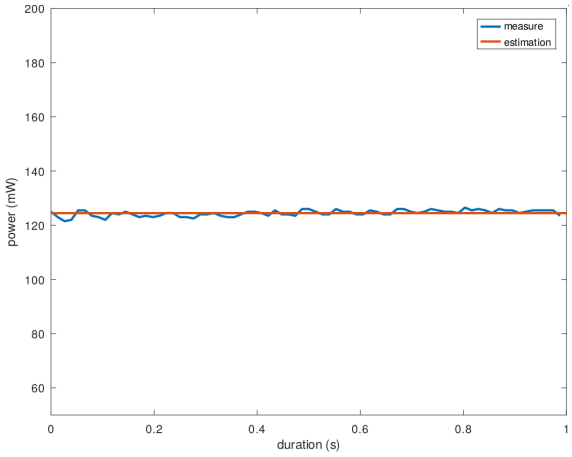
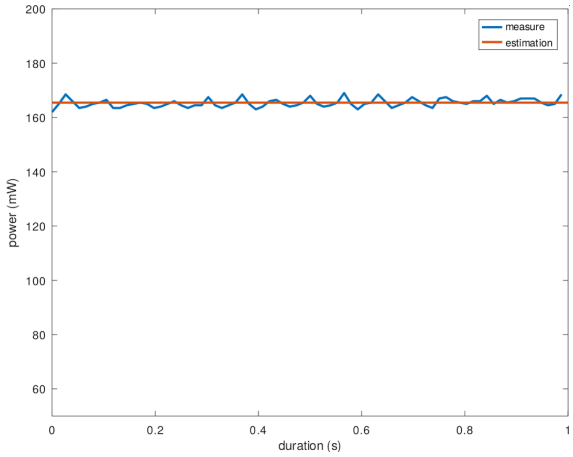


Figure 9. EEL Meta-model

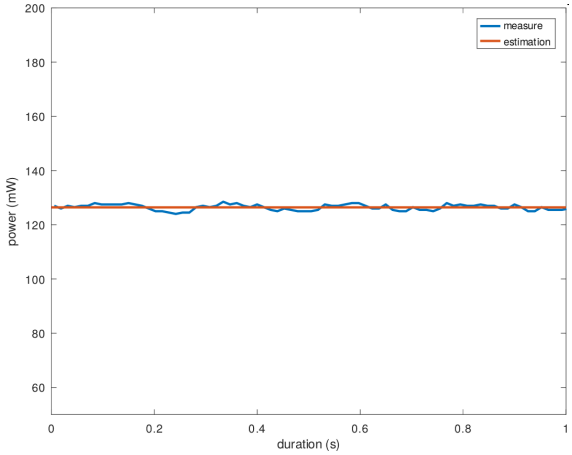
B ArduinoML Benchmarks



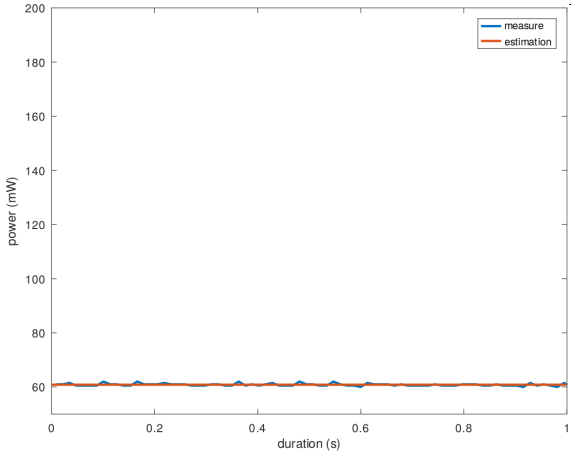
(a) Arduino UnoR3 running with no module.



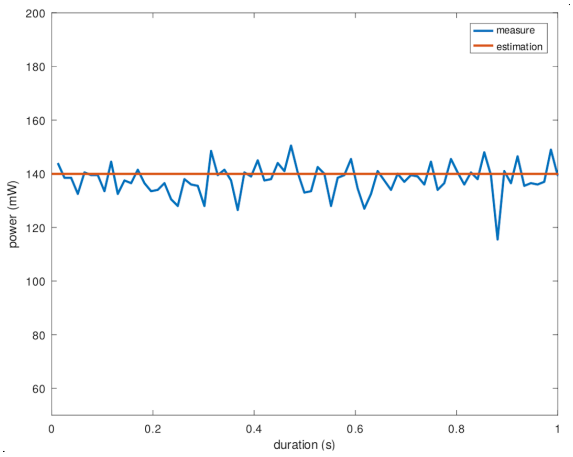
(b) Arduino UnoR3 running with LED on.



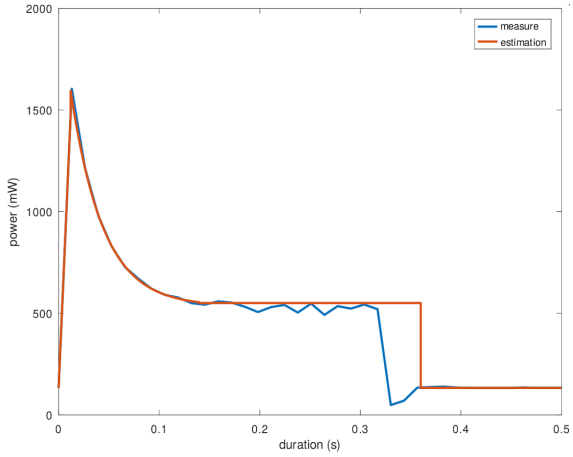
(c) Arduino UnoR3 running with Infrared sensor.



(d) Arduino UnoR3 sleeping with Infrared sensor.



(e) Arduino UnoR3 running with Photoresistor.



(f) Arduino UnoR3 running with Servo Motor rotating 180°.

Figure 10. ArduinoML Benchmarks Used for Building Arduino UnoR3 EEM

References

[1] Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. 2014. The Power of System Call Traces: Predicting the Software Energy Consumption Impact of Changes. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering* (Markham, Ontario, Canada) *(CASCON '14)*. IBM Corp., USA, 219–233.

[2] Deniz Akdur, Vahid Garousi, and Onur Demirörs. 2018. A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture* 91 (2018), 62 – 82. https://doi.org/10.1016/j.sysarc.2018.09.007

[3] Rabie Ben Atitallah, Smail Niar, Alain Greiner, Samy Meftali, and Jean Luc Dekeyser. 2006. Estimating energy consumption for an MPSoC architectural exploration. In *International Conference on Architecture of Computing Systems*. Springer, 298–310.

[4] Nils Bandener, Christian Soltenborn, and Gregor Engels. 2010. Extending DMM behavior specifications for visual execution and debugging. In *International Conference on Software Language Engineering*. Springer.

[5] Olivier Barais, Benoit Combemale, and Andreas Wortmann. 2017. Language Engineering with the GEMOC Studio.

[6] Luca Berardinelli, Antinisca Di Marco, Stefano Pace, Luigi Pomante, and Walter Tiberti. 2015. Energy consumption analysis and design of energy-aware WSN agents in fUML. In *European Conference on Modelling Foundations and Applications*. Springer, 1–17.

[7] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2013. Powerapi: A software library to monitor the energy consumed at the process-level. *ERCIM News* 2013, 92 (2013).

[8] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution framework of the gemoc studio. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 84–89.

[9] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient debugging for executable DSLs. *Journal of Systems and Software* 137 (2018), 261–288.

[10] Erwan Bousse, Tanja Mayerhofer, and Manuel Wimmer. 2017. Domain-Level Debugging for Compiled DSLs with the GEMOC Studio.

[11] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. 2010. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 173–174.

[12] Joseph Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. 2001. Ptolemy: A framework for simulating and prototyping heterogeneous systems. In *Readings in hardware/software co-design*. 527–543.

[13] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2017. *Investigating the Energy Impact of Android Smells*. Technical Report 10. https://hal.inria.fr/hal-01403485/file/carette-saner-17.pdf

[14] Eric Cariou, Olivier Le Goaer, Léa Brunschwig, and Franck Barbier. 2018. A generic solution for weaving business code into executable models.. In *MODELS Workshops*. 251–256.

[15] Maxime Colmant, Mascha Kurpicz, Pascal Felber, Loïc Huertas, Romain Rouvoy, and Anita Sobe. 2015. Process-level power estimation in vm-based systems. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–14.

[16] Benoît Combemale, Xavier Crégut, and Marc Pantel. 2012. A design pattern to build executable DSMLs and associated V&V tools. In *2012 19th Asia-Pacific Software Engineering Conference*, Vol. 1. IEEE.

[17] Robertas Damaševičius, Vytautas Štuikys, and Jevgenijus Toldinas. 2013. Methods for measurement of energy consumption in mobile devices. *Metrology and measurement systems* 20, 3 (2013), 419–430.

[18] Juan De Lara and Hans Vangheluwe. 2002. AToM 3: A Tool for Multi-formalism and Meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 174–188.

[19] Brian Dougherty, Jules White, and Douglas C Schmidt. 2012. Model-driven auto-scaling of green cloud computing infrastructure. *Future Generation Computer Systems* 28, 2 (2012), 371–378.

[20] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. 2012. Xbase: implementing domain-specific languages for Java. *ACM SIGPLAN Notices* 48, 3 (2012), 112–121.

[21] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. 2000. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *International Conference on the Unified Modeling Language*. Springer, 323–337.

[22] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 307–309.

[23] Taher Ahmed Ghaleb. 2019. Software energy measurement at different levels of granularity. In *2019 International Conference on Computer and Information Sciences (ICCIS)*. IEEE, 1–6.

[24] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. 2015. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*.

[25] OM Group et al. 2012. Structured metrics metamodel (smm). *no. October* (2012), 1–110.

[26] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. 2013. Estimating mobile application energy consumption using program analysis. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 92–101.

[27] Cécile Hardebolle and Frédéric Boulanger. 2007. Modhel'x: A component-oriented approach to multi-formalism modeling. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 247–258.

[28] Ábel Hegedüs, István Ráth, and Dániel Varró. 2012. Replaying execution trace models for dynamic modeling languages. *Periodica Polytechnica Electrical Engineering and Computer Science* 56, 3 (2012).

[29] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. 2015. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling* 14, 2 (2015), 905–920.

[30] Zachary King, Mohammed Sayagh, and Abram Hindle. 2016. Energy Profiles of Java Collections Classes. (2016).

[31] Thibault Béziers La Fosse, Massimo Tisi, Erwan Bousse, Jean-Marie Mottu, and Gerson Sunyé. 2019. Towards platform specific energy estimation for executable domain-specific modeling languages. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 314–317.

[32] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. 2013. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 78–89.

[33] Xun Li, Pablo J Ortiz, Jeffrey Browne, Diana Franklin, John Y Oliver, Roland Geyer, Yuanyuan Zhou, and Frederic T Chong. 2010. Smartphone evolution and reuse: Establishing a more sustainable model. In *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 476–484.

[34] Kenan Liu, Gustavo Pinto, and Yu David Liu. 2015. Data-oriented characterization of application-level energy optimization. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 316–331.

[35] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. 2013. xMOF: Executable DSMLs based on fUML. In *International Conference on Software Language Engineering*. Springer, 56–75.

[36] Tanja Mayerhofer, Manuel Wimmer, Loli Burgueño, and Antonio Vallecillo. 2018. *Specifying quantities in software models*. Technical Report. Submitted.

[37] David Mosteller, Michael Haustermann, Daniel Moldt, and Dennis Schmitz. 2019. Integrated Simulation of Domain-Specific Modeling Languages with Petri Net-Based Transformational Semantics. In *Transactions on Petri Nets and Other Models of Concurrency XIV*. Springer, 101–125.

[38] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. 2006. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf* 2 (2006), 157.

[39] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. 2012. A preliminary study of the impact of software engineering on GreenIT. In *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*. 21–27. https://doi.org/10.1109/GREENS.2012.6224251

[40] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2013. A review of energy measurement approaches. *ACM SIGOPS Operating Systems Review* 47, 3 (2013), 42–49.

[41] MF da S Oliveira, Lisane B de Brisolara, Luigi Carro, and Flávio Rech Wagner. 2006. Early embedded software design space exploration using UML-based estimation. In *Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*. IEEE, 24–32.

[42] Marcio FS Oliveira, Eduardo W Brião, Francisco A Nascimento, and Flávio R Wagner. 2007. Model driven engineering for MPSOC design space exploration. In *Proceedings of the 20th annual conference on Integrated circuits and systems design*. 81–86.

[43] Pablo J Ortiz, Jeffrey Browne, Diana Franklin, John Y Oliver, Roland Geyer, Yuanyuan Zhou, and Frederic T Chong. 2015. Smartphone Evolution and Reuse : Establishing a More Sustainable Model Smartphone Evolution and Reuse : Establishing a more Sustainable Model. 90 (2015). https://doi.org/10.1109/ICPPW.2010.70

[44] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (2016), 83–89. https://doi.org/10.1109/MS.2015.83

[45] Rui Pereira, João Saraiva, Haslab Inesc Tec, Nova Lincs, and João Paulo Fernandes. 2016. The Influence of the Java Collection Framework on Overall. (2016).

[46] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 22–31.

[47] Giuseppe Procaccianti, Héctor Fernández, and Patricia Lago. 2016. Empirical evaluation of two best practices for energy-efficient software development. *Journal of Systems and Software* 117 (2016), 185–198. https://doi.org/10.1016/j.jss.2016.02.035

[48] Charles Reams. 2012. *Modelling energy efficiency for computation*. Ph.D. Dissertation. University of Cambridge.

[49] Felix Rieger and Christoph Bockisch. 2017. Survey of approaches for assessing software energy consumption. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*. 19–24.

[50] Felix Rieger and Christoph Bockisch. 2020. Evaluating Techniques for Method-Exact Energy Measurements: Towards a Framework for Platform-Independent Code-Level Energy Measurements. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 125–128. https://doi.org/10.1145/3341105.3374105

[51] Eric Saxe. 2010. Power-efficient software. *Commun. ACM* 53, 2 (2010), 44–48.

[52] Jed Scaramella. 2007. Solutions for the Datacenter ' s Thermal Challenges. January (2007).

[53] Gayane Sedrakyan and Monique Snoeck. 2016. Enriching Model Execution with Feedback to Support Testing of Semantic Conformance between Models and Requirements. (2016).

[54] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. 2016. *United States Data Center Energy Usage Report*. Technical Report. http://eta-publications.lbl.gov/sites/default/files/lbnl-1005775{_}v2.pdf

[55] Digvijay Singh and William J Kaiser. 2010. The atom LEAP platform for energy-efficient embedded computing. (2010).

[56] Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. 2014. Formalizing execution semantics of UML profiles with fUML models. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 133–148.

[57] Chris Thompson, Jules White, Brian Dougherty, and Douglas C Schmidt. 2009. Optimizing mobile application performance with model–driven engineering. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 36–46.

[58] Chiraz Trabelsi, Rabie Ben Atitallah, Samy Meftali, Jean-Luc Dekeyser, and Abderrazek Jemai. 2011. A model-driven approach for hybrid power estimation in embedded systems design. *EURASIP Journal on Embedded Systems* 2011 (2011), 1–15.

[59] Molly Webb et al. 2008. Smart 2020: Enabling the low carbon economy in the information age. *The Climate Group. London* 1, 1 (2008), 1–1.