Model-driven methods for dynamic analysis applied to energy-aware software engineering

Thibault Beziers La Fosse

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de doctorat de

L'École Nationale Supérieure Mines-Télécom Atlantique
Bretagne Pays de la Loire - IMT Atlantique

École Doctorale N°601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

Thibault BÉZIERS LA FOSSE

Model-driven Methods for Dynamic Analysis applied to Energy-Aware Software Engineering

Thèse présentée et soutenue à Nantes, le 29 Janvier 2021
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes
Thèse N° : 2021IMTA0232

Composition du Jury

Président :	**Antoine Beugnard**	Professor, IMT Atlantique
Rapporteurs :	**Ana Cavalli**	Professor, Télécom SudParis
	Romain Rouvoy	Professor, University of Lille
Examinateurs :	**Sébastien Mosser**	Professor, Université du Quebec à Montréal
	Gustavo Pinto	Associate Professor, Federal University of Pará
Directeur de thèse :	**Gerson Sunyé**	Associate Professor, HDR, Université de Nantes
Co-encadrants de thèse :	**Massimo Tisi**	Associate Professor, IMT Atlantique
	Jérôme Rocheteau	Associate Professor, ICAM Nantes
Invité :	**Jean-Marie Mottu**	Associate Professor, Université de Nantes

Acknowledgments

I would like to thank all the members of my jury: Ana Cavalli, Romain Rouvoy, Antoine Beugnard, Sébastien Mosser, and Gustavo Pinto, for reviewing my work, attending my defense, and for the interesting questions and discussions that followed my thesis defense.

Furthermore, I would like to sincerely thank my thesis advisors: Gerson Sunyé, Massimo Tisi, Jérôme Rocheteau, and Jean-Marie Mottu for their guidance, support, and patience that helped maintaining my work in the right way.

I also thank all the NaoMod team members for their constructive feedback during our team meetings and coffee breaks, with a particular thought for my office mates Joachim and Jolan that made A246 the best room to work in. Kudos to our stolen sofa.

Finally I would like to thank my friends and family that were behind me during this thesis, always supportive even during the hard times. Thank you all for the joyful moments, for the long board game nights, for the trips to Etel. I wouldn't have succeeded without you.

Dedication

To Andréas
To Charline
To my Grandma, you would have been so proud

Résumé Français

Contexte

La modélisation est une activité récurrente dans tous les domaines scientifiques et d'ingénierie qui consiste à représenter chacune des facettes d'un système à un niveau d'abstraction adapté. Ces représentations sont appelées "modèles", et permettent une meilleure compréhension de concepts complexes, si bien que les modèles sont présents dans de nombreux domaines: mathématiques, biologie, ingénierie, etc... En informatique, les modèles sont largement utilisés pour décrire tous les aspects d'un logiciel: architecture, algorithmes, communications entre ses composants, et son matériel.

L'Ingénierie Dirigée par les Modèles (IDM) est un processus de développement qui centralise l'utilisation des modèles dans toutes les phases de la création d'un logiciel: design, génération de code, vérification et validation, maintenance, évolution, etc... Dans le contexte de l'IDM, il est commun de vérifier qu'un logiciel est conforme dès sa phase de conception en effectuant différents types d'analyses sur le modèle à partir duquel le code sera généré.

De telles analyses peuvent être faites de deux manières: *statiquement*, ou *dynamiquement*. Les analyses statiques sont effectuées en examinant le modèle sans l'exécuter, par opposition aux analyses dynamiques qui sont effectuées pendant l'exécution du modèle, ou après l'exécution du modèle, par le biais de *traces d'exécution*. Néanmoins, cela implique de rendre le modèle *exécutable*. Ce peut-être effectué en associant une *sémantique d'exécution* au langage de modélisation, cette dernière définissant la manière dont le modèle peut être exécuté.

Énoncé du problème

Si l'IDM est un outil puissant pour concevoir des applications logicielles, une fois l'application déployée et en fonctionnement, il est compliqué de retravailler sur le modèle afin d'améliorer les parties inefficaces du programme. Ainsi, il est important de pouvoir améliorer certains aspects de la qualité du logiciel au plus tôt dans son processus de développement. L'analyse statique de modèles est déjà communément effectuée afin de localiser d'éventuels points d'amélioration, néanmoins effectuer des

analyses dynamiques au niveau du modèle est plus complexe. Effectivement, l'IDM est majoritairement utilisée durant la phase de conception et dans un objectif final de génération de code. Ainsi peu d'aspects dynamiques sont disponibles au sein du modèle. Néanmoins ce manque d'aspects dynamiques dès la phase de conception freine la détection d'anomalies, qui ne seront donc détectables qu'à l'exécution.

Ce problème est particulièrement présent dans le domaine de l'optimisation énergétique de logiciels. Développer des programmes efficaces énergétiquement est une tâche complexe, qui requiert des connaissances sur la consommation énergétique des programmes et des systèmes sur lesquels ils seront déployés. Des études portées sur un large panel de développeurs ont montré que la vaste majorité n'ont que peu de connaissances à ce sujet [152], et donc ne considèrent pas l'efficacité énergétique durant la phase de conception. Être capable de prodiguer au développeur des informations sur la consommation énergétique de son système dès la phase de conception pourrait très certainement l'aider à considérer ces aspects, et les améliorer. Ainsi, les challenges adressés dans cette thèse sont les suivants:

1. Effectuer une analyse dynamique sur un modèle de conception requiert que ce modèle puisse représenter le comportement du système conçus pendant son exécution.

2. Dans le cadre de systèmes cyber-physiques, le modèle de conception définit en plus le matériel sur lequel la partie logicielle est déployée. Cette couche de complexité supplémentaire doit être considérée afin d'effectuer des analyses dynamiques.

3. Les modèles de conception doivent pouvoir être utilisés afin de prodiguer au développeur des informations sur la consommation énergétique de son système. Ces informations doivent enfin pouvoir l'aider à comprendre et améliorer cette consommation.

Contributions

Analyse dynamique de programmes Java

Afin de répondre à ces problèmes, plusieurs contributions sont présentées dans cette thèse. Tout d'abord, une approche permettant d'injecter des traces d'exécution au sein de modèles est proposée. Cette approche génère un modèle statique à partir du code source d'un programme. Une étape d'*instrumentation* permet de tracer l'exécution du programme. Ces traces d'exécution sont enfin injectées dans le modèle statique, permettant ainsi de représenter le comportement dynamique du programme.

Tout d'abord, un premier prototype permettant la modélisation de traces d'exécution de programmes Java a été développé[1]. Le logiciel fait appel à MoDisco afin de transformer un programme Java en modèle. MoDisco est un outil d'ingénierie inversée, historiquement développé dans l'équipe NaoMod. MoDisco effectue une lecture du code source, et en génère un modèle. Ce modèle représente le programme Java, à un niveau de granularité très fin, et peut être utilisé pour le transformer, le moderniser, ainsi que le vérifier par le biais de techniques d'ingénierie dirigée par les modèles. Dans cette thèse, nous injectons des traces d'exécution du programme Java dans ce modèle MoDisco, afin de pouvoir effectuer des analyses dynamiques du programme, en travaillant sur le modèle.

Deux cas d'utilisation font ensuite usage de ce modèle statique enrichi d'aspects dynamiques. Un premier cas d'utilisation mesure et attache des consommations énergétiques aux traces d'exécution du modèle dynamique. Une analyse dynamique est effectuée, permettant la mesure des propriété énergétiques du programme Java[2]. Ces propriétés énergétiques sont injectées dans le modèle MoDisco, aux côtés des traces d'exécution, afin de permettre la transformation du modèle dans une optique d'amélioration de l'efficacité énergétique du programme. Ces propriétés énergétiques sont modélisées avec le standard SMM, utilisé dans le cadre du projet Européen MEASURE finançant cette thèse.

Un deuxième cas d'utilisation fait usage du modèle dynamique dans le contexte d'analyse des impacts que les modifications du code source peuvent avoir sur les tests de régression. Les tests de régression vérifient, après modification du code source, que le programme modifié fonctionne toujours comme attendu par les développeurs. En utilisant le modèle du code source enrichi avec des traces d'exécution, il est possible de ne sélectionner et exécuter que les tests effectivement impactés par les modifications du code source, à la place de l'intégralité des tests. L'évaluation effectuée pour cette contribution analyse des dépots sur GitHub[3]. Un fichier de configuration peut être passé en paramètre de notre logiciel, définissant une liste de dépôts GitHub à analyser. Les dépôts seront alors téléchargés, et toutes les révisions, dans une portée définie en amont, seront considérés. Ce procédé simule un contexte d'intégration continue, où des modifications du code source sont introduites unes à unes. A chaque modification, notre prototype détermine quelles parties du code source sont impactés, et vérifie si ce code impacté perturbe le fonctionnement du dépôt analysé. Cette *sélection de tests de régression* allège effectivement le coût de l'étape de vérification.

La seconde contribution de cette thèse permet l'analyse dynamique de systèmes cyber-physiques (SCPs), par le biais d'une plate-forme développée par l'ICAM de

[1] https://github.com/atlanmod/dynamicanalyser
[2] https://github.com/atlanmod/energymodel
[3] https://github.com/atlanmod/mde4rts

Nantes, nommée Emit. Un langage de modélisation est proposé afin de modéliser des SCPs. Ensuite, une transformation de modèle automatise la surveillance du SCP modélisé dans Emit. Emit peut ainsi monitorer l'exécution du SCP, et en permettre son analyse dynamique.

La dernière contribution présentée introduit Eel, un langage générique permettant de modéliser des formules mathématique pour l'estimation énergétique[4]. Ces estimations sont attachées à des langages exécutables. La consommation énergétique des modèles définis avec ces langages exécutables peut être calculée en évaluant les formules spécifiées avec Eel. En utilisant Eel pendant la phase de conception, un développeur peut connaître la consommation énergétique de son programme avant même d'en générer le code. Cela lui permet de pouvoir optimiser son programme et d'en réduire l'empreinte carbone. L'implémentation de Eel est intégrée dans GEMOC Studio, un environnement de développement de langages et d'exécution de modèles. Eel a été utilisé dans le contexte de l'exécution de modèles Arduino, et est capable d'en estimer la consommation énergétique avec grande précision (95.1%, en moyenne). Ces outils sont open-source, et disponibles sur la plate-forme GitHub.

Contexte de la thèse

Cette thèse a été financée par une co-tutelle entre l'Institut Catholique des Arts et Métiers (ICAM), et l'Institut Mines Télécom (IMT), dans le cadre d'un projet Européen ITEA-3 intitulé MEASURE[5]. Le but de ce projet est d'améliorer la qualité et l'efficacité de l'ingénierie logicielle, tout en réduisant les coups et le temps de production. Ce projet implémente une plate-forme comprenant de nombreux outils de qualité logicielle[6]. Cette plate-forme permet la collection, l'analyse, le stockage et la visualisation de mesures logicielles, définies avec le standard de l'Object Management Group (OMG) : Structured Metrics Meta-model (SMM).

De plus, ce thèse a pris place au sein de l'équipe NaoMod (anciennement Atlanmod). NaoMod est une équipe du Laboratoire des Sciences du Numérique de Nantes (LS2N), localisée sur les campus de l'UFR Sciences et Techniques, et l'IMT Atlantique de Nantes. L'équipe est spécialisée dans l'IDM dans la région de Nantes depuis les années 1990, et a proposé de nombreuses technologies, notamment basées sur Eclipse, a destination des développeurs et architectes logiciels, dans l'optique d'améliorer leur productivité, ainsi que la qualité des applications qu'ils développent[7].

[4] https://github.com/atlanmod/eel
[5] https://itea3.org/project/measure.html
[6] https://github.com/ITEA3-Measure/MeasurePlatform
[7] [8]

Contents

Chapter 1

Context

1.1 Introduction

Modeling is a recurrent activity is all scientific disciplines which aims at representing particular parts of the real world in a simplified manner. As they are able to ease the understanding of complex concepts, models are used in many domains: mathematics, biology, civil engineering, software product lines, or philosophic disciplines [7, 10, 12, 65, 156]. In the domain of information technologies, models are widely used to describe all the aspects of softwares and systems under development: architectures, algorithms, communications, hardware, or components. To unify the language for defining models, the Object Management Group (OMG) defined the standard Unified Modeling Language (UML) [169].

Model-Driven Engineering (MDE) is a process that promotes models as the central key element for all phases in a software development lifecycle. MDE relies on abstract models instead of standard code-based engineering practices to manage complex software systems, and automate various tasks. Over the last decade, MDE has emerged as a successful and a widely used approach for developing software systems. Relying on standard modeling languages (e.g., UML) enables a better interoperability between all the development phases. A typical MDE approach first models a software architecture. The model is verified, validated, corrected if needed, and finally the software can be generated through a model transformation. Furthermore, MDE is also an efficient approach for refactoring and modernizing existing software systems, using model-driven reverse engineering.

Improving the quality of a software, either at design time or runtime, can be done by performing analysis on the model it is designed with. An analysis can be done by examining the model without executing it, i.e., *statically*. For instance, asserting that the class names conform with the naming standards. In contrast with *static* analysis, *dynamic* analysis is performed on models during their execution, or a posteriori, using *execution traces*.

A model alone is simply an abstract representation of *something*. To run a model, an execution environment has to understand how the elements in the model should behave according to the context. This be done by associating *semantics* to all the elements defined by the modeling language. Generally, this is done in two ways: either generating executable code and running it, or directly interpreting the model. We indicate a modeling language that can be executed as a xDSL (executable Domain Specific Language).

As an example, counting the number of entities of a model that are called during an execution requires a dynamic analysis. Furthermore, instead of performing dynamic analysis *while* a model is being executed, such analysis can also be done on *execution traces*. An execution trace is a sequence that contains all the relevant information about a model's execution, over time. It records all the states of the model's execution, the user inputs, the steps responsible for the changes in the

model, the duration of each state, and any other information related to the model's execution.

Energy efficiency is an important concern when designing software, for ecological, economical, or technical reasons. Measuring the energy that a software consumes may require this software to be executed, and dynamically analyzed. Information about the energy consumption of the software is valuable, as it can help improving inefficient software artifacts. Providing energy-related information as early as possible, e.g., at design time, especially using models, would certainly help towards that purpose. This is also beneficial in the domain of cyber-physical systems, where software can be distributed over complex networks of energy-constrained devices.

1.2 Problem Statement

MDE is a powerful method for designing and generating software code. If static analysis of models is already efficient and widely used, performing dynamic analysis, especially at design-time, is more complex. In fact, as design models are used for code generation, little dynamic aspects are available in them before. If this lack of dynamic aspects in software models is an impediment to more efficient implementations, it is also a recurrent issue when designing the underlying physical systems. Thus, limited possibilities of dynamic analysis in models prevents early improvements of software, systems, and the combination of both in the domain of cyber-physical systems.

This is especially true in the area of energy efficiency. Developing energy efficient applications requires a vast knowledge about energy consumption of software and systems, and often requires complex dynamic analysis and tooling, that most developers do not have. Providing feedback to the developers about the energy consumption of their systems, early in the design process, would certainly help them improving the energy efficiency their applications.

Thus, leveraging MDE for a better energy efficiency can be summarized by the following challenges, addressed in this thesis:

1. Performing dynamic analysis on a model requires this model to be executable. When a model is executed via code generation, the execution trace is distinct from the model. Being able to inject an execution trace of a running program into the model it has been generated from can hence enable dynamic analysis, at the model level.

2. In the context of cyber-physical systems, the behavior of an executable model can be distributed over complex networks. Tracing the execution of such system requires complex monitoring platforms. Generating a monitoring

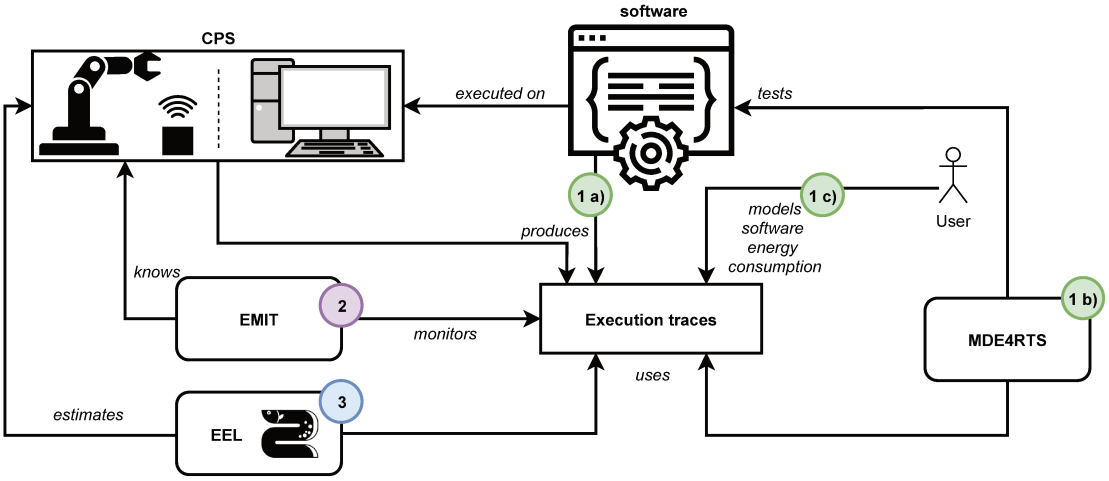


Figure 1.1: Contributions of this thesis.

platform from models of the system would promote an easier access to its execution traces, and enables dynamic analysis.

3. Models of software and systems have to be used to provide relevant energy-related information to the developer. Using that information about energy consumption, the developer can improve the implementation of its program and the design of its system.

If a model driven approach for providing early energy consumption feedback to a developer can target a single language, providing such feedback regardless of the language would foster a better reusability of this work.

1.3 Contributions

A first part of this thesis is focused on modeling execution traces of running software and systems inside static models. A static model of this kind, enhanced with the dynamic aspects of a program, can then provide a support for energy concepts that a developer would need to improve her application. A second part of this thesis focuses on the traceability of CPSs, and tackles the complexity of CPS monitoring. The last part of this thesis aims at generalizing instruction-based energy estimations to any xDSL. An energy-estimation DSL is proposed, that, when applied to execution traces, estimates the energy consumption of any executable model.

Figure 1.1 presents the contributions of this thesis, labeled from 1 to 3, how they are centered around the concept of modeled execution traces.

The first contribution, labeled 1a) presents an approach for modeling execution traces of Java programs inside models. A reverse engineering step creates a model out of a Java source code. An instrumentation phase traces the execution of the program, and injects into the model the traces produced. Two usages of this model are proposed in contributions 1b) and 1c).

A first application, 1b), relies on these execution traces to perform change impact analysis. When changes in the source code are detected, the execution trace can be used to highlight the parts of the code that are impacted by the changes. Then, a *Regression Test Selection* operation is performed, which consists in only verifying the code that is actually impacted, instead of the entirety of the program, hence lightening the testing costs. A second application, 1c), attaches energy consumptions to the element of this model, and using the execution trace, estimates the energy consumption of the model's semantic operations.

The second contribution, labeled 2), presents Emit, a monitoring platforms for cyber-physical systems (CPS)s. A CPS is first designed using a DSL, and successive transformation steps configures Emit in order to automatize the monitoring of the CPS. Emit traces all the events happening in the system, and enables dynamic analysis over the CPS.

The third contribution, labeled 3), presents Eel (Energy Estimation Language). Eel is a generic DSL that enables the specification of energy estimation functions. Energy-related concepts can be attached to the semantic of any executable language. When models written with this language are executed, the energy-related concepts modeled with Eel can be evaluated, in order to estimate energy consumptions. Eel is meant to be used at design time, in order to provide early feedbacks to the developer, so that she can improve the energy consumption of her software.

1.4 Outline of the thesis

This thesis is organized as follow: Chapter 2 presents some background material and related work, used along this thesis. Chapter 3 presents our approach for dynamic analysis in models, separated in three sections: section 3.2 focuses the injection of execution traces inside source code models, section 3.3 use this model of execution traces for energy modeling, and section 3.4 performs regression test selection with it. Chapter 4 presents our approach for CPS monitoring and dynamic analysis, and Chapter 5 presents our language for energy estimation of executable models.

1.5 Scientific Production

During this thesis, we produced 6 articles: 2 international conferences, 3 international workshops, and 1 journal.

— International journal

1. **Béziers la Fosse, T.**, Mottu, J.M., Tisi, M. and Sunyé, G. Source-Code Level Regression Test Selection: the Model-Driven Way. In The Journal of Object Technology (JOT), 2019.

— International conferences

1. **Béziers la Fosse, T.**, Mottu, J.M., Tisi, M. and Sunyé, G. Annotating Executable DSLs with Energy Estimation Formulas. In Software and Language Engineering (SLE), 2020.

2. **Béziers la Fosse, T.**, Rocheteau, J., Cheng, Z., Mottu, J.Model-Driven Engineering of Monitoring Application for Sensors and Actuators Networks. In Software Engineering and Advanced Applications (SEAA), 2020.

— International workshops

1. **Béziers la Fosse, T.**, Tisi, M., Bousse, E., Mottu J.M., and Sunyé, G. 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) (pp. 314-317). IEEE.

2. **Béziers la Fosse, T.**, Mottu, J.M., Tisi, M., Rocheteau, J., and Sunyé, G. Characterizing a Source Code Model with Energy Measurements. In Workshop on Measurement and Metrics for Green and Sustainable Software Systems (MeGSuS), 2018.

3. **Béziers la Fosse, T.**, Tisi, M., and Mottu, J.M. Injecting Execution Traces into a Model-driven Framework for Program Analysis. In : Federation of International Conferences on Software Technologies: Applications and Foundations. Springer, Cham, 2017. p. 3-13.

Chapter 2

Background

In this chapter we introduce the main concepts that this thesis relies on, as well as related work.

Section 2.1 provides an overlook of the Model-Driven Engineering concepts. This includes the core concepts of models, meta-models and model transformations, followed by executable modeling, and a final focus on the standard models proposed by the Object Management Group, used in this thesis. Furthermore, it introduces languages and meta-models used along this thesis.

Section 2.2 gives a bird's eye view on software analysis techniques. It first describes existing static software analysis techniques, followed by dynamic software analysis techniques, as well as the software tooling that is necessary to perform them. It finally describes a specific kind of software analysis: change impact analysis, and its main usage in this thesis: regression test selection.

Finally Section 2.3 focuses on energy efficiency. It presents energy measurement and estimation tooling, and existing software-based approaches for fostering better energy efficiency.

2.1 Model-Driven Engineering

In the early 2000s, the Object Management Group (OMG) introduced the concept of Model-Driven Architecture (MDA). The main purpose of MDA is to provide users with tooling for solving the issues that complex systems could raise. Towards that purpose, an approach to specify IT systems is defined, which separates the system functionalities from their implementation on a specific platform, and supply "vendor-neutral interoperability specifications". MDA promotes the usage of Platform-Independent Models (PIM) as primary artifacts to design system and software architectures. These models can then be adapted to a specific platform (i.e., Platform-Specific Model (PSM)) using successive transformations, refinements, and finally, code generation [20, 97]. We call this process *forward engineering*. On the contrary, PIM can be reconstructed by analyzing PSM. This new PIM can then be used as a basis for code modernization, maintenance and enhancement. We call this process *reverse engineering* [41].

Model-Driven Engineering (MDE), is a broader concept that includes MDA. It is not limited to architecture, but also processes and analysis. MDE has been successfully used in the past decade, and has proven itself to be a powerful asset during all stages of software lifecycle: early design, modernization, analysis, refactoring. Existing studies reported benefits in terms of quality, productivity and maintainability compared to traditional development processes [90]. For these reasons, MDE is getting gradually adopted by companies looking for efficient methodologies for software engineering, such as Ericsson, Volvo, Thales, or ATOS,

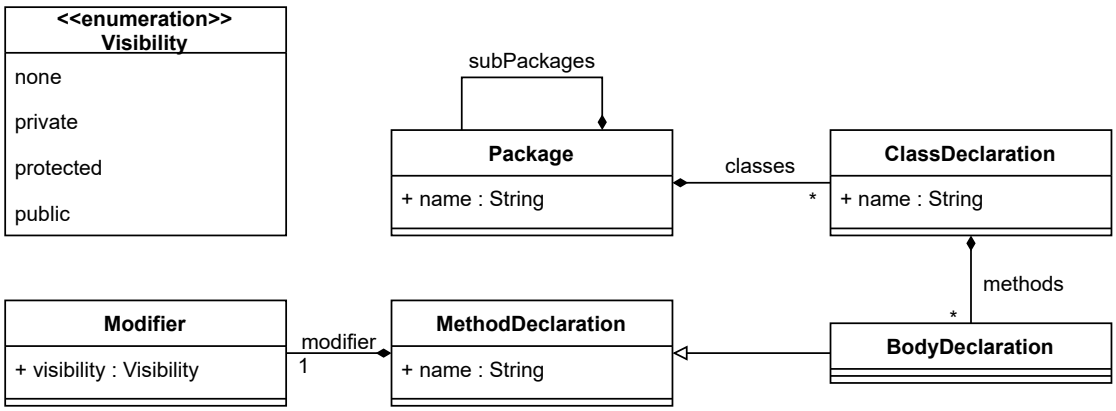


Figure 2.1: Java meta-model

and a quick glance at any job search website shows how popular MDE tools are [199].

2.1.1 Models, Meta-Models and Languages

Meta-models

A meta-model is the explicit specification of an abstraction [21, 126], and is strongly related to the concept of ontology [78]. All the meta-models used in this thesis are defined using the OMG's Meta-Object Facility (MOF)[1]. A meta-model identifies a list of concepts (*classes*, each being composed of *properties*), with *relationships* between them, and *semantic rules* regulating how the model can be denoted, and that must be satisfied by conforming models.

As an example, we introduce a meta-model, the Java meta-model, used in this thesis [49]. This meta-model defines all the entities that can be written with the Java language. *Package* represents the Java packages, root containers of the classes. *Class* corresponds to the Java classes, and can contain *Methods*, or other *Class*, as the inner classes of Java programs. Each *Method* contains a *Block* of *Statements*. These *Statements* can be either *ControlStatement*, representing Java `if`, `while` etc ... Or *ExpressionStatement* and *InvocationStatement*. The former contains one or multiple *Expressions*, which can be any expression that Java could define (mathematical, boolean, etc...), whereas the later represent the invocation of a Java method. We show an excerpt of this Java meta-model in Figure 2.1.

Furthermore, we introduce a second example, also used later in this thesis: the meta-model of ArduinoML. ArduinoML is a *Domain Specific Language* (DSL) used to represent the structural and behavioral aspects of Arduino systems. Thus this

[1] https://www.omg.org/mof/

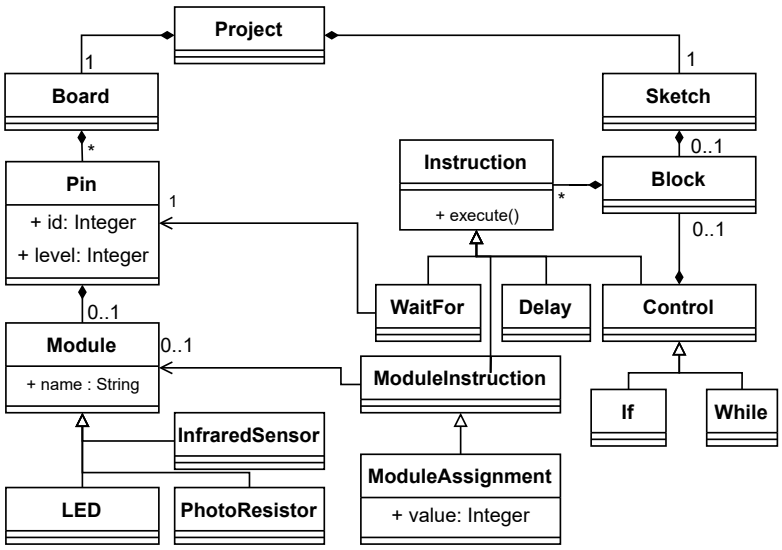


Figure 2.2: Abstract syntax of Arduino Modeling Language

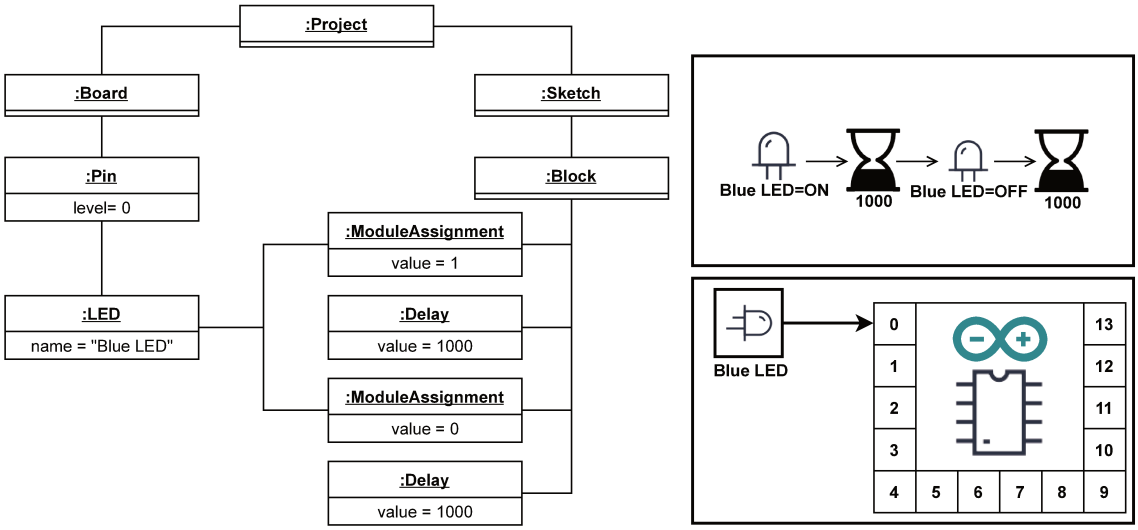
meta-model defines ArduinoML's *abstract syntax*, defining the concepts that are expressed with the language and the relations between them. Figure 2.2 shows an excerpt of this meta-model. The root element of this meta-model is the *Project* class. It has a name to identify it, and contains a *Board*, describing the structural aspects of a Arduino-based system, and a *Sketch*, describing the behavioral aspects of the Arduino system. The *Board* is composed of *Pins*, on which *Modules* are plugged. Pins hold a signal, defined by the *value* attribute, and are identified by an integer. *Modules* are any kind of sensors and actuators that can be used in a Arduino system, such as *LEDs*, *Photo-resistors* or *Infrared sensors*, and are plugged to pins. The *Sketch* meta-class is composed of a *Block*. This *Block* contains a list of *Instructions*. Each *Instruction* can either be a *Control* instruction, in order to define loops and branches, a *Delay*, in order to add timers in the Arduino's behavior, or module-related instructions. Those *ModuleInstruction* can be used in order to fetch the state of a module, or assign a value to it.

Models

The concept of *model* can be defined w.r.t. the concept of instance in the class-based object oriented paradigm. If a class can be instantiated into an object, then a meta-model can be instantiated into a model, that conforms to the meta-model abstract syntax and satisfies the meta-model semantics.

Figure 2.3a provides an example of a ArduinoML model, instance of the ArduinoML meta-model presented in Figure 2.2. It conforms to the abstract syntax of the Arduino meta-model, and respects its semantics. This model describes a

Arduino board, on which a single LED is plugged. This LED is plugged to the Arduino pin labeled "0", and the initial signal sent from the Arduino, to this LED, is 0, meaning that the LED starts turned off. The sketch object of this model defines the behavior of this Arduino board. It consists of a list of instructions to represent the blinking of the LED. The first instruction assigns a low (i.e., 0) signal to the LED, that turns it off (if the LED was on). This instruction is followed by a 1 second Delay, then a new assignment turns the LED on, and a final one second delay concludes the behavior defined in this model. Thus this model defines a Arduino system with a blinking LED.



(a) Abstract syntax representation of an ArduinoML model

(b) Concrete syntax representation of an ArduinoML model

Figure 2.3: Two representations of the same ArduinoML model.

Languages

A common manner of defining models is through the usage of a *Language*. Two categories of languages can be considered: *General Purpose Languages* (GPL), and *Domain Specific Languages* (DSL). GPLs are meant to define models in a wide variety of application domains. Thus, they do not include languages meant to be used in a specific application domain. Common programming languages, such as Java or C++ are considered as GPLs.

On the contrary, DSLs define particular abstractions dedicated to a specific discipline. A DSL is composed of an *abstract syntax*, defining the concepts that are expressed with the language and the relations between them. This abstract syntax is defined through the means of a meta-model. Furthermore it has a *concrete*

syntax, i.e., a human-readable way of using the language, that can be a *textual syntax* or *graphical syntax*. Textual syntax enables editing the model using a textual notation. As an example, the CSS and HTML markup languages can be considered as DSLs, and are most of the time defined using their corresponding textual syntaxes. Graphical syntax, on the contrary, is meant to define the model using a graphical interface. For instance the Scratch programming language is used through a visual block-based graphical syntax. Finally, a language has *semantics*, defining the meaning of the concepts expressed.

As mentioned earlier, ArduinoML is a DSL, with an abstract syntax, defined by the meta-model in Figure 2.2. It also has a concrete graphical syntax, defined with Sirius [196], and Figure 2.3b shows the concrete graphical representation for defining the model in Figure 2.3a.

2.1.2 Model transformations

Model transformations are among the most widely used concepts in MDE. It is defined as an automatic operation, that consists of the production of one or more models, from one or more input models. Model transformations are defined at the meta-model level, and applied on models conforming to these meta-models. Model transformations can be defined using any GPL (e.g., Java), or using dedicated DSLs, popular in the MDE community, such as ATL [95] for model-to-model transformations, or Acceleo [131] in the context of model-to-text transformations. To standardize model transformations, the OMG introduced the QVT (Query/View/Transformation) specification, later implemented in many languages and frameworks [58, 96, 101, 123].

A model transformation can be either *endogenous*, or *exogenous* [124]. The first category define transformations between models defined with the same language. The second category defines transformations between models conforming to different meta-models. Code generation is an example of exogenous model transformation, that we use in the context of ArduinoML. This transformation is implemented using Acceleo, and generates C code to be deployed on the Arduino board. The C code corresponding to the Arduino model presented in Figure 5.6 is shown in Figure 2.4. This is an example of *forward engineering*: an abstract, high-level, representation of the system is first defined through a model, which is then used to produce the low-level concept, the C source code.

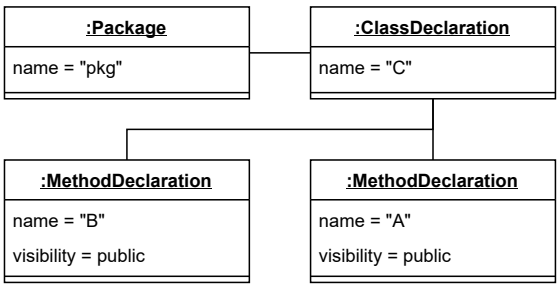
In contrast, *reverse engineering* generates this high-level abstract model of a concept from its low-level representation. The MoDisco framework performs this text-to-model transformation. It takes a syntactically correct Java source code, and transforms it to a model, conforming to the Java meta-model presented in Figure 2.1. This model can then be refined, enhanced or verified, and a new version of the source code can be generated with final model transformation.

```
1         void setup() {
2             // initialize digital pin LED_BUILTIN as an output.
3             pinMode(LED_BUILTIN, OUTPUT);
4         }
5
6         // the loop function runs over and over again forever
7         void loop() {
8             digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on (
    HIGH is the voltage level)
9             delay(1000);                        // wait for a second
10            digitalWrite(LED_BUILTIN, LOW);     // turn the LED off by
     making the voltage LOW
11            delay(1000);                        // wait for a second
12        }
13
```

Figure 2.4: C code produced via a model transformation with input model in Figure 2.3b

```
package pkg;
class C {
  public void A() {
    B();
  }
  public void B() {
    System.out
        .println("Hello, World!");
  }
}
```

(a) Java program to reverse engineer



(b) Excerpt of the Java model produced

2.1.3 Executable meta-modeling

If MDE is often used for designing the structural aspects of software and systems, it also enables the definition of behavioral aspects, as shown in Figure 2.3b. Executing models, conforming to a DSL, is the purpose of *executable meta-modeling*. Executing a model requires the DSL it conforms to to provide *execution semantics*. These semantics are rules, attached to the meta-classes of the meta-model's abstract syntax, that describes how to execute the modeled elements. We call *executable DSL (xDSL)* a DSL that can define *executable models*.

Model execution is popular in the domain of verification and validation. In fact, executing a model (i.e., before deploying) is an efficient manner of checking that it behaves as intended by the model developer. Simulators can rely on executable models, and provide useful tools, such as debuggers, to analyze the state of the

```
def void execute() {
    val pin = _self.getPin(_self.module)
    pin.level = _self.value
}
```

Figure 2.5: Transformation rule attached to the ModuleAssignment meta-class, defined with Kermeta3

executable model at runtime.

As stated before, executing a model requires *execution semantics*. In this thesis we will focus on two types of execution semantics: *operational* semantics [155] and *translational* semantics [110]. Both enable the execution of models, but behave differently. First, *operational* semantics are defined as endogenous model transformations that changes the state of the model during its execution. Each operation of the semantics is defined as a set of *steps*, and each *step* is an endogenous model transformation. Second, *translational* semantics consist in an *exogenous* model transformation, that translates the model into a second one. This second model conforms to the abstract syntax of a second xDSL, with the *operational* semantics needed to execute it. If both approaches are effective to run models, tracing the execution of models that conform to a xDSL with *translational* semantics can be harder, as the domain of the language it has been transformed to can differ from the original language.

We previously introduced ArduinoML, a language for defining Arduino systems. ArduinoML comes with *operational* semantics, enabling the execution of conforming models, using a dedicated engine. These semantics defines transformation rules (i.e., *steps*), that are attached to the classes of the abstract syntax of the language. Objects, instances of these classes, can be executed using these rules.

As an example, running the ArduinoML model presented in Figure 2.3b would iterate over all the instructions defined in the *Sketch*, and execute them, one by one. Running the first *ModuleAssignment* would apply the transformation rule presented in Figure 2.5 on the model, and transform it into a the model shown in Figure 2.6. This step assigns the value 1 to the Pin on which the targeted module is plugged.

2.1.4 Meta-modeling Standards

The Object Management Group (OMG) proposed several standard meta-models and languages during the last two decades, in order to tackle recurrent needs in the MDE community. We present three of these technologies in what follows: UML, OCL and SMM.

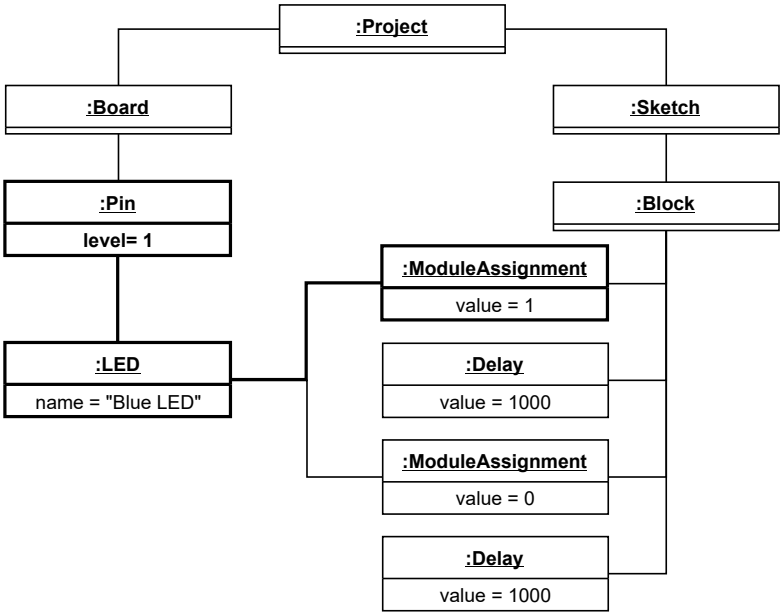


Figure 2.6: Arduino model after running the first *ModuleAssignment*

UML stands for Unified Modeling Language, and was first introduced in 1996 by the OMG. It is the standard language for object oriented modeling. Its purpose is to unify existing modeling technologies. UML proposes 14 types of diagram, to represent all the aspects of software engineering. These types of diagram can be divided in two categories: *Structure*, *Behavior*. Structure diagrams include the *Class* diagram and *Component* diagram, used to model the entities of the system. Behavior diagrams include the *Activity* and *Sequence* diagrams, that represent the business operations that the system performs, as well as the communication between its entities.

UML models can be associated with Object Constraint Language (OCL) expressions, another OMG standard. OCL is a declarative language. It can be used to define invariants, derivation rules, or queries, that, when evaluated, returns specific elements of the model. We mainly rely on OCL in this thesis as a query language, as it can be used to fetch the values of attributes in the model, and compute arithmetic and logic operations on the model.

The last OMG standard we describe is the Structured Metrics Meta-model (SMM) standard [138]. SMM enables the representation of properties, measurements, and entities performing measurements. It introduces vocabulary from the domain of analysis, used in this thesis:

- *Measure*: A method assigning numerical or symbolic values to entities in order to characterize an attribute of the entities.

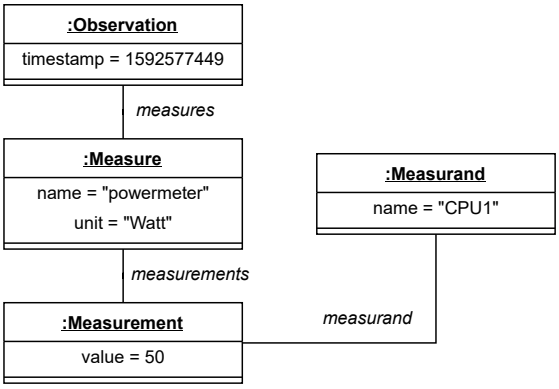


Figure 2.7: SMM model of a power meter measuring the power consumed by a CPU.

- *Measurement*: A numerical or symbolic value assigned to an entity by a *measure*.
- *Measurand*: An entity quantified by a *measurement*.

Figure 2.7 shows a simple use case featuring SMM.

This model shows a power meter device that measures the consumption of a CPU. Thus, the power meter device is modeled with the *Measure*. This measure produces a *Measurement*, a "50W" record, when applied to the *Measurand*, the CPU. The root of this model is an *Observation*, which model the timestamp at which the measure started gathering measurements.

SMM is used as a basis for modeling all information gathered through software analysis, either statically or dynamically. The next section presents some related work about software analysis.

2.2 Software analysis

Software analysis is a common step in any software lifecycle. Since the beginning of software programming, it has always been used for many purposes: quality, flexibility, metrics about the programs, understandability, verification and validation, etc. Developing a complex application right is extremely complicated if no effort is spent in design, especially using modeling and analysis. In the early 2000s, at the beginnings of MDE, Daniel Jackson and Martin Rinard claimed that there is a great opportunity of collaboration between abstract models, and code analysis [91]. Indeed, with a straight relationship between the abstract model and the source code, the developer could leverage properties extracted from the models as if they were directly produced out of the source code. Furthermore, attaching

properties produced by software analysis to design models would help organizing, understanding, and re-using these properties.

Software analysis can be divided in two main categories: static analysis, and dynamic analysis. The next sections define and compare those two approaches.

2.2.1 Static analysis

This first category of software analysis is performed without executing the program [200]. Most static analysis approaches are performed on the source code of the software, or on its compiled code. Static analysis do not require the code to be runnable to be performed, and thus can be applied at any state in the development process of a software. It is usually fast, as it does not depend on the complexity of the program to be analysed, nor on its inputs. For these reasons, most modern development environments embed static analyzers, that provides useful information to the developer in order to improve the quality of her programs. Among its possible applications, static analysis can be used for asserting of the correctness of the program using control flow analysis, for measuring cyclomatic complexities, locating unreachable code, for formal verification, or to check if a software conforms with its design model [69, 71, 132].

Software models have proven to be a powerful asset for performing static analysis. Dedicated query languages, such as OCL, can be used on top of models to perform various static analysis. For instance, this is especially interesting when associated to the MoDisco framework: a Java program can be reverse-engineered into a model, and queries can be performed on this model to statically analyze the source code.

If static analysis is efficient to capture the program's structure, only little information of its behavior can be obtained. If when it comes to behavior analysis dynamic approaches tend to be more efficient and accurate. However, dynamic analysis approaches can make good use of static information to enable a better understanding of the control flows of the system [75].

2.2.2 Dynamic analysis

In contrast with static analysis, dynamic analysis is performed by executing a program, and gathering metrics at runtime. Dynamic analysis is a common approach in software engineering, especially used in the context of verification and validation. Popular usages include test code coverage, fault localization, or any kind of performance analysis [184, 188, 192]. Most dynamic analysis approaches rely on a step of *instrumentation*. In fact, the desired analysis requires some software tweaking in order to be performed.

As an example, classic code coverage approaches require to mark the lines of code that have been executed by the test cases to locate non-tested sections of

code. However simply running the test cases available in the program's source code is not enough: by itself, the program do not know which lines of code are executed. Thus, it is necessary to inject additional behavior inside the program. *Instrumenting* a program to enable code coverage consists in adding *probes* before each line of existing code. When a probe is executed, then its associated line of code is considered as executed. Furthermore, the same approach can be applied to trace the execution of a software. Three main types of instrumentation exist that we detail next.

Instrumentation

Source-code instrumentation

This first instrumentation technique consists in adding new statements inside the existing source-code. Source-code instrumentation is easier to perform than other instrumentation techniques, as it does not have to handle the compiled code but only the initial source code, which is humanly readable. Furthermore, it is very accurate, as all the information (statements, methods, attributes, etc ...) are available in the source code. However source-code instrumentation tends to be slower than other approaches: the source code has to be parsed, rewritten, compiled, and loaded before running it. Figure 2.8 shows a simple example of the source code instrumentation of a Java program. A class is defined with two methods, `A()` and `B()`. The `A()` method simply calls the `B()`one. The source code instrumentation defined injects a statement before and after each method invocation. These statements print, in the standard output of the Java Virtual Machine, the name of the method before and after executing it.

Several source-code transformation approaches have been described in the literature [66, 146, 195]. In this thesis, we rely on the Spoon Framework [148].

Spoon relies on a Java meta-model for representing the Abstract Syntax Trees (ASTs) of Java programs. It provides a set of pre-processors that enable static analyses and transformations of Java programs, and automatically compiles the instrumented code for an immediate execution.

Static binary instrumentation

This second instrumentation technique works at the compiled code level. Instead of injecting additional statements in the source code of a program, it adds compiled code inside the existing binaries of a runnable application [105, 133]. The modified binaries can then be executed.

When users want to reason on source-code instructions, instrumenting at the binaries level is less accurate than at the source code level. Once the code is compiled, information about the initial position of statements in the source code is lost, and only the line numbers are kept. If a very accurate instrumentation method

```
class C {
    void A() {
        B();
    }
    void B() {
        System.out.println("Hello, World! ");
    }
}
```

(a) Java source code to instrument

```
class C {
    void A() {
        System.out.println("before C.B");
        B();
        System.out.println("after C.B");
    }
    void B() {
        System.out.println("before System.out.println");
        System.out.println("Hello, World! ");
        System.out.println("after System.out.println");
    }
}
```

(b) Java source code after instrumenting a method invocation

Figure 2.8: Source code instrumentation of a Java program

is needed, then source code instrumentation is a better choice. Else, in terms of performances and usability, on-the-fly instrumentation is better: it requires the same knowledge about compiled binaries than static binary instrumentation, and does not need to write and load the instrumented binaries to execute them.

Figure 2.9 shows an example of a Java byte code instrumentation. Figure 2.9a shows the compiled code of a Java class, named `C`. The `C()` method first described is the default constructor of this class. The method instrumented is the method labeled `A()`, which consists in a single invocation of the method `B()`. The additional behavior injected through a byte code instrumentation prints information in the standard output before and after calling the method `B()`. If this byte-code instrumentation performs exactly as the source code information previously defined, it requires specific knowledge about the compiled code and some middleware tweaking. Various frameworks have be developed for static byte code instrumentation and are used for both industrial and academic purposes [22, 48, 120, 191], but are outclassed in terms of performances, by on-the-fly instrumentation.

On-the-fly instrumentation

This last instrumentation technique, also called as dynamic byte-code instru-

```
class C {
    C();
        Code:
            0: aload_0
            1: invokespecial #1      // Method java/lang/Object."<init>":()V
            4: return
    void A();
        Code:
            0: aload_0
            1: invokevirtual #2      // Method B:()V
            4: return
}
```

(a) Java byte code to instrument

```
class C {
    C();
        Code:
            0: aload_0
            1: invokespecial #1      // Method java/lang/Object."<init>":()V
            4: return

    void A();
        Code:
            0: getstatic     #2      // Field
               java/lang/System.out:Ljava/io/PrintStream;
            3: ldc           #3      // String before C.B
            5: invokevirtual #4      // Method
               java/io/PrintStream.println:(Ljava/lang/String;)V
            8: aload_0
            9: invokevirtual #5      // Method B:()V
            12: getstatic     #2     // Field
               java/lang/System.out:Ljava/io/PrintStream;
            15: ldc           #6     // String after C.B
            17: invokevirtual #4     // Method
               java/io/PrintStream.println:(Ljava/lang/String;)V
            20: return
}
```

(b) Java byte code after instrumenting a method invocation

Figure 2.9: Byte code instrumentation of a Java program

mentation, consists in rewriting the binaries of a program while it is running [34, 117, 188]. Compared to its static alternative, it does not require the engine (e.g., the Java Virtual Machine in the context of Java), to load the modified binaries to be effective. On-the-fly instrumentation is faster to perform, and usually consists in injecting agents at the start of a program to transform it. The most popular Java frameworks for such instrumentation are ASM [33] and Byte buddy [201].

All these instrumentation approaches are suitable candidates for performing dynamic analysis of software. In this thesis, we rely on software instrumentation for two purposes: tracing software execution, and measuring energy consumption at runtime. We detail the energy-efficiency domain later.

Model-driven dynamic analysis

Most models capture the static representation of softwares and/or systems, and thus little dynamic aspects are available in them. One of the most popular usage of MDE is about the design of software (e.g., using UML class diagrams), eventually using code generation in the end to produce the source code. Typically, no execution is performed with the model but using the generated code instead, and thus no dynamic analysis can be performed in this context. However, in the context of *xDSLs*, models are executable, and hence can be dynamically analyzed. Existing approaches have been proposed for enabling dynamic analysis of *executable* models.

The GEMOC language and modeling workbench propose extension mechanisms that enables the modification of the execution engine. Additional behavior can be defined and executed before and after running the transformation rules [28, 85]. Furthermore, Eric Cariou et al. propose an approach for weaving business code into executable models [40]. Business operations are associated with executable elements, and executed before, during, or after the execution of the targeted element. These approaches are close to what *instrumentation* offers to software dynamic analysis. Finally, additional related work has been proposed to provide feedback to the user during the execution of models, through graphical views in simulation and execution environments [129, 175].

These model driven approaches can be used to perform dynamic analysis in the context of xDSLs, as well as efficiently trace down model executions, and gather *execution traces*. In what follows, we detail execution traces.

2.2.3 Execution traces

We define an execution trace as a sequence that contains all the relevant information about a model's execution, over time. It records all the states of the model's execution, the user inputs, the steps responsible for the changes in the model, the duration of each state, and any other information related to the model's execution. Whilst dynamic analysis can be performed *at runtime*, it can also be performed at any time using an execution trace, either by analyzing the trace, or by re-executing the model using a replay mechanism based on the trace [86].

Tracing the execution of software is extremely common during the software development life-cycle, and has been heavily studied in the literature. Several surveys and studies show that tracing the execution of a software is used in many fields of software engineering, such code coverage [188], fault localization [202], or change impact analysis [111].

The most popular, and used approach to trace the execution of a software is to rely on usual instrumentation techniques. Several approaches instrument the source code or the binaries of a program, and build execution traces in the form of

call graphs [9, 55, 106]. These call graphs serve as execution traces, and are often used in the domain of *impact analysis* to compute dependencies between software artifacts [161]. Knowing the impacts a code modification can have on other parts of the program can really improve the developers life, and reduce the maintenance costs. We detail impact analysis in what follows.

2.2.4 Impact analysis

Impact analysis approaches can be divided in two main categories, static and dynamic dependency collection. If both approaches are individually efficient, dynamic analysis can improve the precision of the impacts compared to static techniques. Thus both approaches are often combined.

Chianti [162] is a dynamic change impact analysis approach based on call graphs. Using two different versions of the source code, Chianti creates a set of *atomic changes*, and analyses their dependencies in order to determine the tests affected by code changes. Basically, considering two atomic changes A_1 and A_2, if adding A_1 in the first version of the source code leads to a program syntactically invalid, then it has a dependency to A_2.

Several static impact analysis approaches have been developed. They mainly parse the Java classes, and extract dependencies at various granularity levels (statements, methods, classes). When the source code is updated, the dependency graph is analyzed and the artifacts impacted are highlighted [36, 67, 84, 93]. An other interesting static approach for impact analysis is to rely on *program slicing* [1, 14, 47, 159, 190]. Program slicing consists into computing a set of points, such as statements, that can have an effect on other point of the program.

Nonetheless this approach suffers from the disadvantages of a static approach: a loss of precision to limit the execution time when the program's source code is being too big and/or complex to analyze. To tackle this issue, impact analysis based on dynamic slicing have been proposed too [193, 197]. Computing the impact that changes can have on the program can help limiting the development costs, especially through *Regression Test Selection* (RTS). This approach consists in only running the test cases that have been impacted by newly introduced changes in the source code, instead of running the entire test suite, to check that no regression has been added. When efficiently performed, RTS effectively reduces the duration of the regression testing phase. We describe in details RTS, as it is an effective asset for software sustainability.

2.2.5 Regression test selection

RTS has been extensively studied, leading to multiple systematic studies and surveys [23, 62, 76, 109, 205]. These approaches for RTS can be classified according

to their granularity. The granularity of a RTS approach is the minimal size of the artifacts at which change impacts are considered. As an example, a RTS approach is considered as *fine-grained* when the impacts are computed at the statement level, whereas it is *coarse-grained* when the impacts are computed at the class level.

Several popular RTS frameworks are coarse-grained, considering file or class granularity. They result in less overhead than a finer-grained one, and tends to be more scalable for bigger systems. Ekstazi has a 3-phases approach: test selection using file-dependencies, test execution, and new dependencies collection [70]. It builds call graphs using a fast on-the-fly instrumentation. Ekstazi is safe for both code changes and file-system changes. It is among the most efficient state-of-the-art RTS solutions, with an average time reduction of 47 %. HyRTS is an hybrid RTS technique, combining both method granularity and file granularity [208]. Using both the cost-effectiveness of file granularity, and the accuracy of the method-level RTS, HyRTS outperforms Ekstazi and other state-of-the-art RTS techniques.

Fine-grained RTS techniques tend to select less tests cases to execute, since the impact analysis is more accurate [141]. This results in a faster test execution, but counter-balanced by a longer impact analysis. As an example, FaultTracer is a RTS tool using Chianti's change impact analysis for a method level test selection [209]. Despite showing accurate results, FaultTracer produces a high overhead, and thus computing the impact analysis is sometimes longer than executing all the test cases.

The dynamic approaches presented are based on software execution traces. Capturing software execution traces can be done on the system currently running the software using instrumentation. However, in the context of cyber-physical systems (CPS), the software is distributed over complex networks running a wide variety of operating systems. This make any type of instrumentation impossible, and dynamic analysis must be performed in dedicated CPS monitoring platforms.

2.2.6 CPS monitoring

In CPS, hardware entities communicate with an information infrastructure, and are orchestrated for certain tasks, such as home automation, car automation, e-health, smart cities, and "Industry 4.0" [4, 13, 52, 102, 147].

In realistic scenarios, most CPSs are too complex to manage without software support. The majority of monitoring platforms for CPSs consist of web services enabling the management of the entities in the network [181] (configuration, connection, disconnection, etc...). Two types of TCP-based protocols are generally used to support communication within the CPS. The first protocol is the HTTP protocol for enabling communication between the applications of the CPS via the request-response messaging paradigm [50]. The second protocol is the Message Queuing Telemetry Transport (MQTT) protocol to support communications from

devices to applications via the publish-subscribe messaging paradigm (widely used between gateways and platforms within IoT architectures). It is lightweight, mature, and require little amount of code to be functional. For those reasons it has been heavily studied and used, in both industrial and research worlds [89, 115, 186].

With its low latency, small energy consumption, and simplicity of use, MQTT is an interesting asset for monitoring and tracing the execution of CPS. Energy consumption is especially important in the realm of CPSs. In fact, these systems often embed small computation units relying on limited power supplies. Being able to design energy-efficient software to deploy on CPS, and estimate the energy consumption of the CPS components, is thus important. In the next section we present the domains of energy efficiency, measurement, and estimation.

2.3 Energy efficiency

Energy consumption has become an important concern in the domain of software engineering during the past decade [144]. The massive electricity needs of data-centers represented 1.8% total U.S. consumption in 2016, and an estimated account for about 2% of global greenhouse gas [45, 177, 198]. Furthermore, the expense to power and cool the data-centers escalated to a significant cost factor: for every $1.00 spent on new hardware, an additional $0.50 is spent on power and cooling [174]. As an example, considering the fastest computer in the world in June 2020[2], the Supercomputer Fugaku, with 7,299,072 cores and peaking at 513,854.7 TFlop/s, consumes 28,335 kW. This would cost 5001.13€ per hour, with a cost of 0.1765€/kWh (average cost in France, on November 2020). As an answer to these alarming numbers, significant efforts aim at reducing the CO_2 emissions of data-centers [173], reducing electricity costs [178, 198], or improving the battery life of smartphones [114, 142].

Energy consumption E is defined as an accumulation of power consumption P over a duration t, such as $E = P \times t$. Thus, reducing the energy consumption of a software can be either be done by: (1) Reducing the duration it takes to execute and/or (2) Reducing its power consumption. Traditionally, such concerns tend to be addressed at the lower levels (hardware or middleware), and several strategies have been presented in the past years towards that purpose [125]. Hardware level optimizations are usually invisible to the developers, and most recent systems include many of them, such as dynamic voltage, frequency scaling, cpu throttling or clock gating [107, 108, 203]. However, studies show that the design of softwares can have a significant impact on energy consumption [30, 38]. If a lot of work from the domains of high-performance computing and complexity can be applied to energy-efficiency, it is not always the case, and sometimes reducing the duration

[2]https://www.top500.org/lists/top500/2020/06/

to execute a program increases the power consumption so much that the energy consumed still increases. As an example, if parallel computing often improves the performances of a software, each CPU core used increases the power consumption of the system, and can result hence a loss of energy-efficiency [153].

Nevertheless, promising results have been introduced using software-level energy management approaches [116], which, combined with lower level optimizations, can effectively reduce the energy consumption of programs. However, if energy-efficiency is an important topic, other studies showed that most developers lack knowledge about energy, as well as tools for measurement and optimization [144, 151]. For these reasons, it is crucial to provide more tools to developers in order to help them coding energy-efficient applications. This implies (1) providing information about the energy consumption of their softwares through energy measurement or estimation and (2) proposing energy-efficient software constructs and design patterns.

2.3.1 Energy measurements

Providing to the developer feedback about the energy consumption of its software can be done either through *measurement* or *estimation*. Measurement techniques differ in granularity, and can be performed at various level in the development environment. In the following, we detail the main approaches for measuring energy consumption.

Power-meters

Power-meters are external devices that can be plugged directly on the power supply of a computer, to measure its entire energy consumption, or on specific components that need to be monitored [87, 88, 207]. The main advantages of power meters is that they do not need any software modification to be used, only a little hardware tweaking and finally their presence to not impact the energy consumption of the measured system. However, the metrics gathered are coarse grained as they usually include the entire system's energy consumption. Thus, understanding the energy consumption of a single program, when the entire system's energy consumption is measured can be complicated. Furthermore, as power-meters are remote devices, synchronizing the timestamps of these metrics with the observed software require additional analysis.

Specialized systems for energy monitoring

Specialized systems for energy monitoring are specifically designed for energy-aware development. Such systems embed multiple sensors, plugged to several parts of

```
class C {
    void A() {
        double beginning = EnergyCheck.statCheck();
        B();
        double energy = EnergyCheck.statCheck() - beginning;
        System.out.println("B consumed "+energy);
    }
    void B() {
        System.out.println("inside method B");
    }
}
```

Figure 2.10: Source code instrumentation using jRAPL to measure the energy consumption of a method call.

the system (e.g., CPU, disks, RAM), and can perform fine-grained measurements at high frequencies. The most popular ones are the Atom Leap [180] and the Spartan Fpga [170]. These tools are extremely accurate when performing energy measurements, however they run specific operating systems and have limited capabilities, and thus cannot be easily used by developers.

Application level energy measurement tools

This last category of energy measurement tools can be used from software and operating system levels. They are usually libraries or registries that can be queried for information about the system's energy consumption. Their main benefits are accuracy, fine-grain, and availability on most operating systems and hardware. However, monitoring the energy consumption of a software at runtime often requires the usage of instrumentation techniques. In fact, when the developer needs energy consumption information about specific sections of her program, she needs to modify the behavior of this program, to enable calls to application-level energy measurement libraries.

As an example we consider the jRAPL energy measurement library [116]. jRAPL is a Java library that enables access, at the JVM level, to RAPL, a Intel feature providing information about the energy consumption of the CPU. jRAPL can be called by through a simple invocation of a Java method, and returns the energy consumed, by the CPU, since the system has been started up. In order to measure the energy consumed by a method with jRAPL, an instrumentation step could add jRAPL invocations before and after this method execution. Comparing the measures obtained hence produces insights of the CPU consumption while running a program, as shown in Figure 2.10.

JRAPL can be used through a single line of Java code, and provides the energy

consumed by the CPU cores. However, the energy it measures includes the energy consumed by the system, and induces a considerable overhead [206], threatening the validity of the measurements.

2.3.2 Energy estimation

While several tools and application are able to provide accurate measures of the energy consumption, many approaches focus on estimating this consumption instead. Estimating an energy consumption often requires less tooling than performing measurement, and is, de facto, easier to setup. A classic way of estimating an energy consumption can be done by estimating a power consumption, and using it along with a duration to estimate an energy consumption. This can be done, as an example, with PowerAPI [25], which estimates the power consumed by the CPU, for a given process, using the CPU usage. Furthermore, PowerAPI proposes a middleware toolkit for defining software-level power meters, and several other energy and power estimation frameworks from the state of the art rely on it [43, 137]. Compared to jRAPL, these frameworks rely on power models and are not impacted by the system consumption.

Lots of effort have been spent in developing efficient application-level monitoring tools, researchers have also proposed instruction-level estimation approaches. These approaches usually attach small energy consumption values to the instructions of a specific targeted language. Each time an instruction is executed, an small energy consumption can be added to the application total consumption. Instruction-level energy estimation approaches can be very accurate, but are complex to set up as the energy consumption of a single instruction can be hard to acquire in the first place.

Using these energy estimation approaches, it is possible to provide to the developers insights of the energy consumptions of their programs. This energy feedback could help them doing better design choices, aimed towards a better energy efficiency. We describe in the next section existing software-level approaches that are efficient at reducing the system's energy consumption.

2.3.3 Energy-aware software engineering

Existing work has shown that software-level optimizations are efficient in reducing the energy consumption of software and thus should be combined with hardware level optimizations. Software-level energy optimizations rely on the developer's implication to be implemented. Many of these optimizations have been described in the state of the art, and can consist in simple code updates, or important design changes. As an example, it can be a simple data structure update. In fact,

existing work compare the energy efficiency of Java data structures according to the context [83, 149].

Another approach for energy efficiency is the *approximate computing* [79]. This approach consists in reducing the quality of service of an application, with a soft error tolerance. The program returns a slightly less qualitative result, but consuming much less energy. It is especially interesting in the context of image processing, where minor changes in the quality of an image are not necessarily visible at first sight.

Many other software-level changes can be performed, depending on the context. For Android mobile programming, approaches have been proposed to optimize the battery duration, such as CPU offloading [104], putting threads on sleep when not used [112], limiting network usage and inputs/outputs [152], reducing the amount of external libraries [112], etc ...

Little work has been done in the domain of MDE for energy-aware software engineering, but some interesting approaches can be mentioned. Several approaches use models to represent system and software architectures, and optimize them for better energy efficiency [57, 103, 139, 140]. This is especially interesting in the domain of cloud energy efficiency, where models can be used to represent cloud systems, and serve as a basis for optimization using evolutionary algorithms and simulation [37]. A few other existing approaches rely on xDSLs. The languages executed embed power and energy estimation features, and can be used to monitor and optimize the systems designed [11, 19, 187, 189].

Finally, frameworks have been proposed to help software engineers doing the best energy-efficient design choices. This is the case of Seeds, which generates several versions of an application implementing different combinations of collections, and picks the most efficient one after monitoring the energy consumption with Leap [119]. Approaches have also been proposed to improve energy efficiency by automatically refactor programs. These approaches define "code smells", "anti patterns" or "bad code constructs", which are known energy consuming code patterns, and automatically fix them [46, 98, 128]. However, they are limited to a small number of situations (e.g., collection implementations), and cannot replace a developer doing the right energy efficient design choices. Thus, it is crucial to help developers writing their program in a more efficient manner.

This chapter presented some background concepts necessary to follow this thesis. It first describes model-driven engineering, first in a general manner, followed by more specific concepts: model transformations and executable models. Then it presents software analysis, along with the tools used in this thesis to perform it. It finally presents the domain of software energy efficiency, with its measurement, estimation, and optimizations tools. The next chapters present our contributions in these areas, adding references to specific related works when necessary.

Chapter 3

Model-driven tracing of software execution

3.1 Introduction

Many properties of a program have to be analyzed during its lifecycle, such as correctness, robustness, energy consumption or safety. These behaviors can be analyzed either dynamically by executing the program, or statically, usually by examining the source code. If these approaches seem to be opposed, they synergize well, as dynamic information can be used to add more precision to static analysis [63, 73].

MDE is an efficient approach to design, modernize, generate and analyse software. However, in most MDE approaches, software systems are engineered by first designing models, refining them until the architecture is satisfying enough for generating the source code. With such workflow it is complicated to perform dynamic analysis on models, as they only represent static aspects of the software, such as the architecture. In this chapter, we want to tackle this problem, by adding dynamic aspects of the software inside the static model it conforms to, to enable dynamic analysis on the model. Towards this purpose, we study the static model-based reverse engineering framework MoDisco, and leverage it to perform dynamic analysis on models.

The MoDisco framework [31] is designed to enable program analysis in MDE by creating a model of the source code, the code model, and using it for program understanding and modernization. The code model makes the program structure easily accessible to external modeling tools for any kind of processing, e.g. for static analysis. MDE tools analyzing source code through MoDisco do not execute the original program. However, if MoDisco provided models of dynamic aspects of the code, this would enable other useful analysis. In what follows, we show a method to build a model of the program execution, using both static and dynamic analysis. Thus, an initial structural model is statically built using MoDisco, containing the basic blocks of a program: packages, classes, methods and statements. Thereafter, the code is instrumented in order to add execution traces to the model during program execution. Consequently, the program is executed and all the statements in the model that have been executed are ordered in an execution trace. Section 3.2 shows this approach. We then provide two useful usages of this model of dynamic aspects of a program serving sustainability purposes.

First, we use it to display and understand the energy consumption of software, as shown in Section 3.3. The execution trace available in the model is enriched with energy consumptions, enabling model-driven energy-aware refactoring of the software.

Then we also use it to lighten the cost of regression testing, as shown in Section 3.4. The execution trace is used as a dependency graph for impact analysis purposes, and the impact that source code changes can have on the test cases can be evaluated with it. Finally, the regression testing phase cost can be lightened by

only running the test cases effectively impacted by changes.

3.2 Modeling software execution traces

3.2.1 Approach

This section illustrates our approach for dynamically building a model of the program execution. We propose an automatic process made of a sequence of three steps. Figure 3.1 gives an overview of this process. On the left-hand side of Figure 3.1, the input is the source code of the considered system. First, a static model is generated thanks to a reverse engineering step. Second, on the left-hand side of Figure 3.1, a source-code instrumentation step prepares the code before execution. Finally, the instrumented code is executed and its instrumentation allows us to complete the analysis model into the dynamic model of the source code. This model is the output of the process, and represent the execution of the program.

The dynamic model should contain the structure of the source code, especially describing the targeted system. Furthermore it should reify which statements are executed when the system is run under the action of a launcher (e.g. a set of tests, or a main method). In addition, the order of the calls should be stored to be used when analyzing the behavior of the system based on the dynamic model.

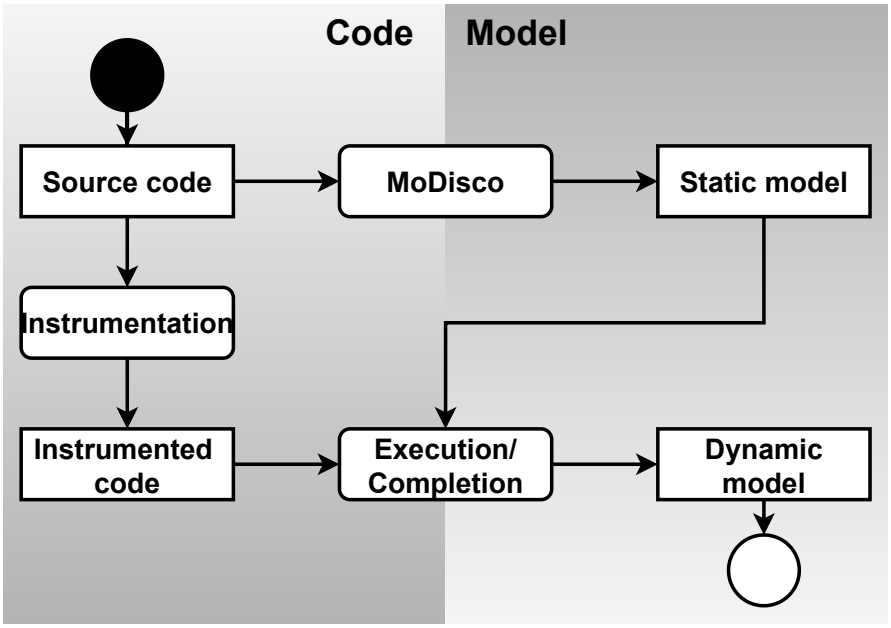


Figure 3.1: 3-step process generating a dynamic analysis model from source code

```
public class C {
  public static void main(String[] args) {
    A();
  }
  public static void A() {
    B();
  }
  public static void B() {
    System.out.println("Hello from method B");
  }
}
```

Figure 3.2: Example of a simple Java program to trace.

In this work, we consider Java source code and we exemplify the approach with the naive class presented in Figure 3.2. This class is the source-code entry of the process in Figure 3.1.

Model Driven Reverse Engineering

The first step of our approach consists of generating the model of the code structure using MoDisco[1]. MoDisco has a visitor-based system which navigates through the full abstract syntax tree (AST), and then builds a model from it, according to its Java meta-model [31]. We use a specific option of MoDisco, which annotates each element of the output model with its location in the source code. This information will be necessary for performing the dynamic analysis, as we show later.

Figure 3.3 shows a simplified version of the model generated by MoDisco from the code in Figure 3.2 (*Static model* in Figure 3.1). *Node* elements contain the position, and a reference to the statements. Since the full static model generated by MoDisco is rather large, we extract the excerpt in Figure 3.3, focusing on statement-level information. For instance we filter out information about expressions, binary files, or import declarations. Since in this work we focus on the execution trace of the statements, only those ones are needed, within their respective classes, methods, and packages containers. Specifically this filtering is required to minimize the final model in-memory size afterwards. The filtering is performed during the model transformation described in the next section.

Once the static MoDisco model is generated, we need to instrument the software to trace its execution. The execution thus captured can then be modelled along with the MoDisco model.

[1] http://www.eclipse.org/MoDisco/

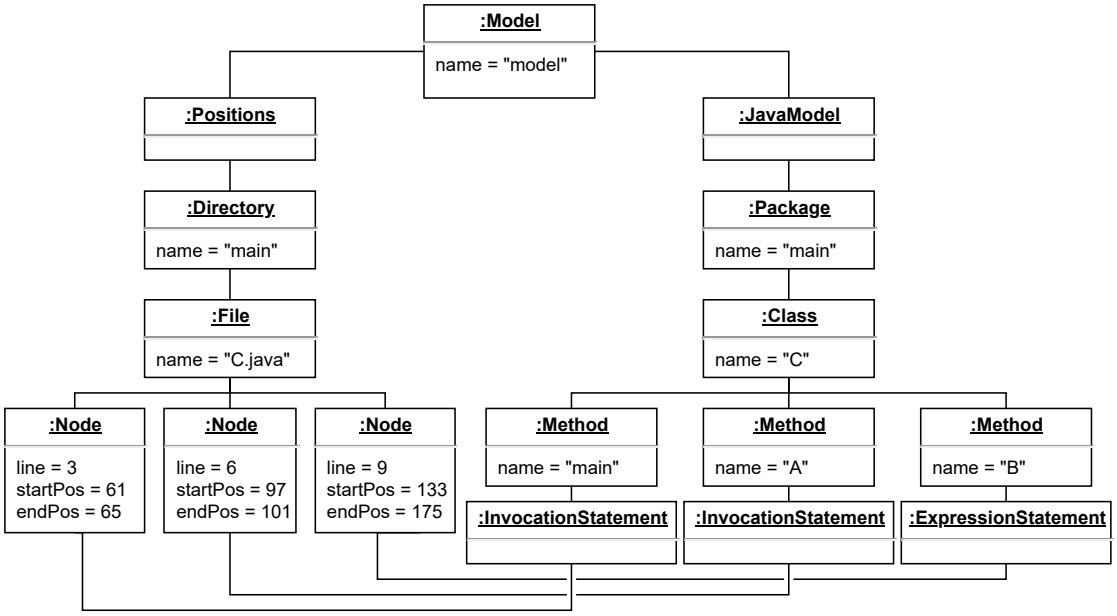


Figure 3.3: MoDisco model of the program in Figure 3.2

Code Instrumentation

Code instrumentation is conducted by inserting additional statements in the code, at specific places, so that when the software is executed, these new statements are also executed [73]. We detail existing instrumentation techniques in Section 2.2.2.

For this approach, we relied on source code instrumentation. Indeed, some specific information about the source cannot be obtained when analyzing the binaries, especially the source code position of the statements. Besides having the line number for each statement like the other approaches, source code instrumentation can also provide column position numbers, a relevant information used to match the executed statements of the code with the analysis model's statements.

The instrumentation has been performed using the Spoon Framework [148]. When instrumenting classes, a `match()` method call is added before each statement. Its parameters are class name where it belongs, the method, and finally the source code position of this statement. The source-code instrumentation generates the code of Figure 3.4, which is compiled and loaded before the execution.

Execution and Injection

Following the example depicted in Figure 3.4, the execution part consists of running the program, through the `main()` method here. When the new instrumented statements are executed, the MoDisco model statically built is completed with

```
public class C {
  public static void main(String[] args) {
    match("C", "main", 61, 65)
    A();
  }
  public static void A() {
    match("C", "A", 97, 101)
    B();
  }
  public static void B() {
    match("C", "B", 133, 175)
    System.out.println("Hello from method B");
  }
}
```

Figure 3.4: Instrumented code tracing the execution of the program.

dynamic information. The method `match()`, iterates over the static model to find the statement being executed, using its qualified class name, method name, and finally the source code position of the statement.

For each statement executed, a *Measurement* is created, conforming to the SMM standard meta-model. In this example, we label this measurement as `"executed"`. Then, `Measurements` are associated with *MeasurementRelationships*, thus reproducing the execution order of the program. Relying on SMM for modelling this trace enables a better flexibility of our approach, as many more dynamic information can thus be gathered at runtime, attached to the elements of the MoDisco model: energy consumption, execution times, etc...

This model dynamically completed represent the execution of the software, as showed in Figure 3.5. Furthermore, statements not linked to any `"executed"` measurements can be considered as not executed, this is an immediate interesting information in the context of code coverage. Additionally, considering a statement that is part of an execution trace, modifying this statement could eventually impact the execution of the program, and thus the impacts of this modification can be estimated simply by looking at the trace. This can serve as a dependency graph, useful in the domain of impact analysis. Finally, this model answers to the problematic announced in this section, as it is dynamically created, and analyze the program's behavior. However this model highly depends on the source code's size. Indeed, each statement in the source code corresponds to one element in the analysis model, thereby an important source code might lead to scalability issues.

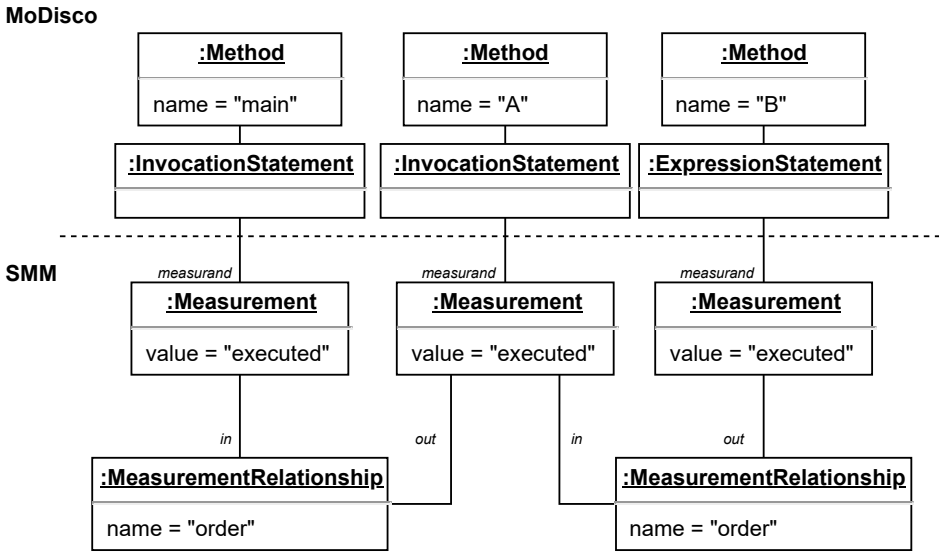


Figure 3.5: Model of the execution trace of the `main()` method execution of Figure 3.2

3.2.2 Evaluation

Execution environment

In this section we evaluate the overall performances of our dynamic model generation framework, using the XMI persistence layer for our models.

This evaluation is conducted on a Java project containing the classes introduced in Section 3.2.1. We programmatically increase the size of this project by duplicating the number of classes, and for each class we propose a test class that runs it. This way, every test class is testing a single target class. Using this setup, we can manage the size of the output model, and observe the behavior of our prototype with either small and big models.

The experiments are executed on a desktop computer running Windows 10 Professional 64Bits, using an Intel Core i7-4770K (3.50GHz) CPU and, a Sandisk SSD PLUS 1TB. The Java Virtual Machine version is JDK 1.8.0_121, and runs with a maximum Java heap size of 2,048 MB.

This experimentation starts by running the program analysis on Java projects containing a few dozen of classes, which can be considered as small here. Subsequently, this number of classes is increased, up to thousands. We measured the execution time for each step of our prototype, and reported it in Figure 3.6. As described in the previous parts, those steps are: Reverse Engineering with MoDisco (RE), Source code Instrumentation with Spoon (Instr), Test Execution (Exec), and Injection of the traces (Inj) .

When the project under analysis reaches approximately 12,000 test classes, the model created by MoDisco using reverse engineering is too big to be stored in memory, thus preventing any other analysis on bigger projects.

Discussion

The curves from the left diagram in Figure 3.6 are showing the growth of execution times when the number of classes increases. The other diagram shows the same data, but its representation gives a better understanding of each step's duration in the whole process and its total duration. This diagram shows that the MoDisco static model generation is by far the longest steps of the program analysis, with a non-linear complexity. Also, as written in the previous subsection, the MoDisco static model creation will not be achieved when the program under analysis gets very big (approximately 12,000 test classes) due to a lack of memory and the well-known XMI scalability problem.

Javier Espinazo-Pagán et al. explained in their paper [143] that the XMI persistence layer scales badly with large models, due to the fact that XMI files cannot be partially loaded. Indeed the XMI resource needs to keep the complete object in the memory to use it.

This scalability problem can be partially resolved using a more scalable persistence layer for the EMF Models, such as NeoEMF [18] or CDO[2]. Nonetheless, MoDisco has its own meta-models, and uses EMF generated code. Using this code with NeoEMF and CDO resources cannot currently improve the scalability, since those layers need to generate their own code from an Ecore meta-model.

[2]https://eclipse.org/cdo/

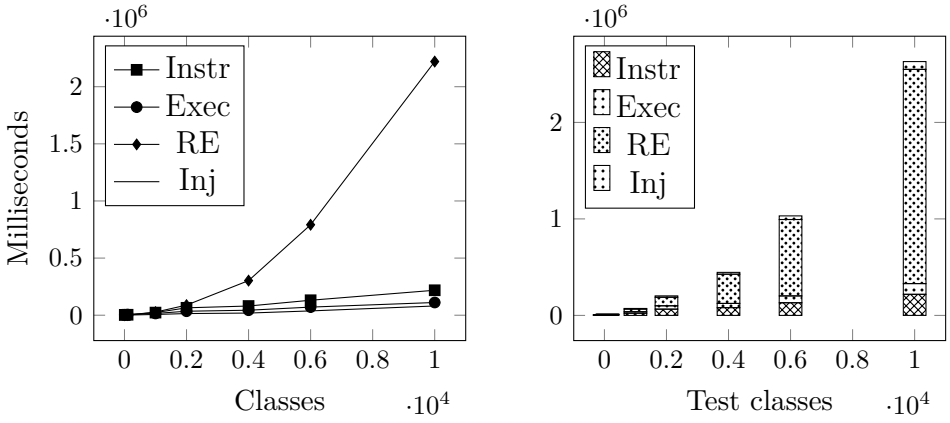


Figure 3.6: Dynamic program analysis execution times using the XMI persistence layer.

3.2.3 Conclusion

In this section we have presented our approach for dynamic program analysis purposes, using Model Driven Engineering. The steps of the process dynamically generate a model, and models an execution trace along with it conforming to the SMM standard. If this approach suffers from the well studied scalability issues of MDE, it also creates relevant models of execution traces, reusable for any kind of software dynamic analysis. The next sections present usages of this trace model to improve the sustainability of software. The first usage relies on this model to represent the energy consumption of the software. A second uses it as a dependency graph to compute the impacts of software changes on the test suites.

3.3 Characterizing the source code model with energy measurements

3.3.1 Introduction

As stated in Section 2.3, energy consumption becomes a major concern when developing software. Traditionally, such concerns tend to be addressed at the lower levels (hardware or middleware), and several strategies have been presented in the past years towards that purpose [125]. Additionally, promising results have been introduced using software-level energy management approaches [116], which, combined with lower level optimizations, can effectively reduce the energy consumption of programs.

Unlike hardware-level energy saving techniques, software-level approaches often need the implication of developers. In fact, to develop a *greener* code, a certain level of energy-awareness is helpful to make relevant design choices. Studies showed that more than 80% of programmers did not consider the energy consumption when developing softwares, even if they know how important it can be, especially in the realm of mobile application development [119, 144]. Representing source code along with energy consumption would help developers making the right design choices.

As stated before, MDE is well-known for improving reusability and flexibility by relying on standard meta-models. Thus, models are typically used during the software development, as specification models to generate a source code, or as source code models, reverse-engineered from the software in order to analyze or refactor it. In this section we extend the MDE approach previously described for modeling the source code of a software and characterizing it with energy measurements. To this extent, we rely on the model in which we injected execution traces, as presented in Section 3.2. A first model of this software is generated with the static reverse-engineering framework MoDisco [32]. Then, on top of the execution trace gathered dynamically, we perform energy measurements that are modeled in the trace.

These measurements are also persisted in a second model conforming to the Structured Metrics Meta-model (SMM). In order to characterize the source code with energy metrics, the two models are associated: elements in the source code (e.g., Java methods) are linked to the energy measurements, enabling the analysis of the source code energy consumption.

This section is organized as follows: Section 3.3.2 presents our approach, Section 3.3.5 discusses it and Section 3.3.6 presents the threats to the validity of our approach, finally Section 3.3.7 concludes the section.

3.3.2 Approach

The approach used here is close to the one described in Section 3.2.1, with a few changes. If the approach presented before focused on traces at the statement level, in this section we study the energy consumption at the method level. Indeed, measuring the energy consumption of statements is extremely complicated, as the duration of a statement execution can be too short to calculate an energy consumption. Since we are coarser-grained, we can rely on an on-the-fly byte-code instrumentation instead, as it is easier and faster to set up.

We detail in this section the several steps necessary to characterize our source code model with energy measurements. First, the source code model is statically built using the MoDisco reverse-engineering framework. Second, the code is instrumented, in order to add probes inside the program, which is then executed to gather the energy measurements at runtime, and trace its execution. Third, the measurements are persisted in a model conforming to SMM, and associated with the MoDisco source code model. The Java program in Figure 3.7 is used to describe our approach. Our application is available on GitHub[3].

Considering a source code written in Java, MoDisco generates the corresponding model conforming to a Java meta-model. The energy measurements have to be dynamically gathered, and are not available in the initial MoDisco model. Nonetheless, a benefit of MDE is that such measurements can be modeled and then associated with our source code model, as presented in the next sections. Applying MoDisco on the program in Figure 3.7 would produce a model containing all the source-code elements of such program. An excerpt of this model is available in Figure 3.8.

3.3.3 Energy Measurements Computation

Once the source code model is generated, the energy is measured. In order to get such measurements for each *method* in the program, it has to be instrumented. We use the ASM on-the-fly instrumentation library[4] for that purpose. The JVM byte-code is visited, and probes are added in the byte-code of every method of the program. A first probe is added in every method entry point, traces down the method call, gets the system time, and finally gets the energy consumed by the CPU at the specific moment, in microJoule.

Another probe is added at every method exit point. This second probe computes the duration of this method execution, and the energy it has consumed. To do so, the energy consumed by the CPU is fetched a second time, and subtracted from

[3]https://github.com/atlanmod/EnergyModel
[4]https://asm.ow2.io/

```
public class App {
  public static void main(String [] args) {
    App app = new App();
    app.methodA ();
    app.methodB ();
  }
  public void methodA () {
    // some lightweight computations
  }
  public void methodB () {
    // some heavy computations
  }
}
```

Figure 3.7: Simple Java program

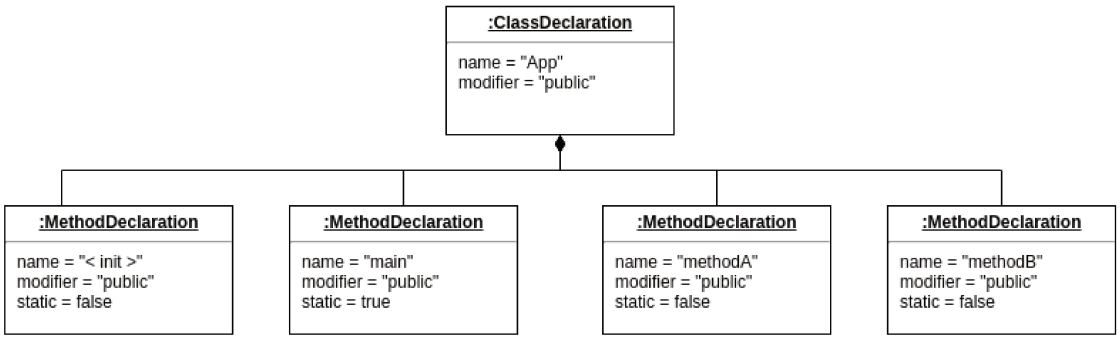


Figure 3.8: Excerpt of the MoDisco source code model

the value obtained in the first probe. Our energy measurements are performed using the same approach as the `jRAPL` framework [116].

As an example, considering the Java source-code available in Figure 3.7, the byte-code generated, and its instrumented version would be the ones in Figure 3.9a and Figure 3.9b respectively. Figure 3.9a contains the byte-code of the `main` as method described in Figure 3.7. This byte-code first initializes the App class, and successively calls the methods `methodA()`, and `methodB()`.

Figure 3.9b calls the methods `methodA()` and `methodB()`, however, probes have been added at the entry and exit points of the method. The probe located at the entry point of the method is called by instantiating a class named `Probe`. The constructor of this class takes as a parameter the qualified name of the method it → MT "is" located in, and computes the energy consumed before the execution. Once the exit point of the method is reached, the `exit` method of the probe is called. This method gets the energy consumed, compares it with the value obtained at the entry point, and traces that value.

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=1
       0: new           #2  // App app
       3: dup
       4: invokespecial #3  // new App()
       7: astore_1
       8: aload_1
       9: invokevirtual #4  // app.methodA()
      12: aload_1
      13: invokevirtual #5  // app.methodB:()
      16: return
```

(a) Java byte-code before instrumentation

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=1
       0: new           #11   // Probe probe
       3: dup
       4: ldc           #25   // String App$main
       6: invokespecial #16   // new Probe()
       9: astore_1
      10: new           #2    // App app
      13: dup
      14: invokespecial #3    // new App()
      17: astore_2
      18: aload_2
      19: invokevirtual #4    // app.methodA()
      22: aload_2
      23: invokevirtual #5    // app.methodB()
      26: aload_1
      27: ldc           #25   // String App$main
      29: invokevirtual #19   // probe.exit()
      32: return
```

(b) Java byte-code after instrumentation

Note that if a second method is called inside a first method, then the energy consumed by this first method includes the energy consumed by the second one. For instance, if we consider the code in Figure 3.7, the energy consumed by the `main()` method will include the energy consumed by the `methodA()` and `methodB` methods. Hence it could be possible to approximate the energy consumed only by `main()` by subtracting the energy `methodA()` and `methodB` consumed, thanks to the execution trace.

Running the instrumented code triggers the probes to gather measurements. For each method, the following metrics are obtained:

1. Its timestamp
2. The energy it consumed

3. The methods that have been called inside.

Running the code can be done either by executing the main method of a program, or through test cases. During the execution, the measurements are published on a messaging queue, in order to be analyzed after the execution. Indeed, analyzing the traces is an expensive operation, and performing it after the execution limits the overhead induced by the execution of the instrumented statements.

Furthermore, our approach is relying on existing energy measurement tools, and do not aim at improving their accuracy. Performing the measures causes a significant workload, limiting the accuracy of the energy measurements [165]. For that reason, we can only provide the energy consumed at the method level, or at coarser granularity. Per-instruction energy consumption is thus out of the scope of this approach, and a different approach.

3.3.4 Energy Measurements Modeling

In this main step of our approach, we attach to the execution trace the energy measurements in a model conforming to SMM and associate it to the source code model that MoDisco generates. This is performed by reading the measurements sent in the messaging queue one by one. Energy consumed, timestamps and internal method calls are persisted as elements in the model. Finally the measurements are linked to the methods from the source-code level, hence characterizing them.

In Figure 3.9, the energy consumed is contained in the `Measurement` elements, as values. The method calls are represented using the `MeasurementRelationship`. The "before" and "after" links point towards the `Measurement` elements, describing in which order the methods have been called. Here for instance, `main()` calls `methodA()` and `methodB()`.

Furthermore, the timestamps of the methods invocations are available in the `ObservedMeasure` elements. Finally the source code model elements (e.g. `MethodDeclaration`) are associated to the `Measurement` using the link labeled as `measurand`.

3.3.5 Discussion

This dynamic model characterized with energy measurements offers developers a better energy-awareness than standard software engineering techniques:

First, a static model of the program is created. The association between MoDisco's Java meta-model and SMM offers a representation of the source code characterized with dynamic information, available for analysis purposes. We rely on this combination of meta-models to propose graphical representations of the programs, embedded as Eclipse plugins. We propose two views defined using the

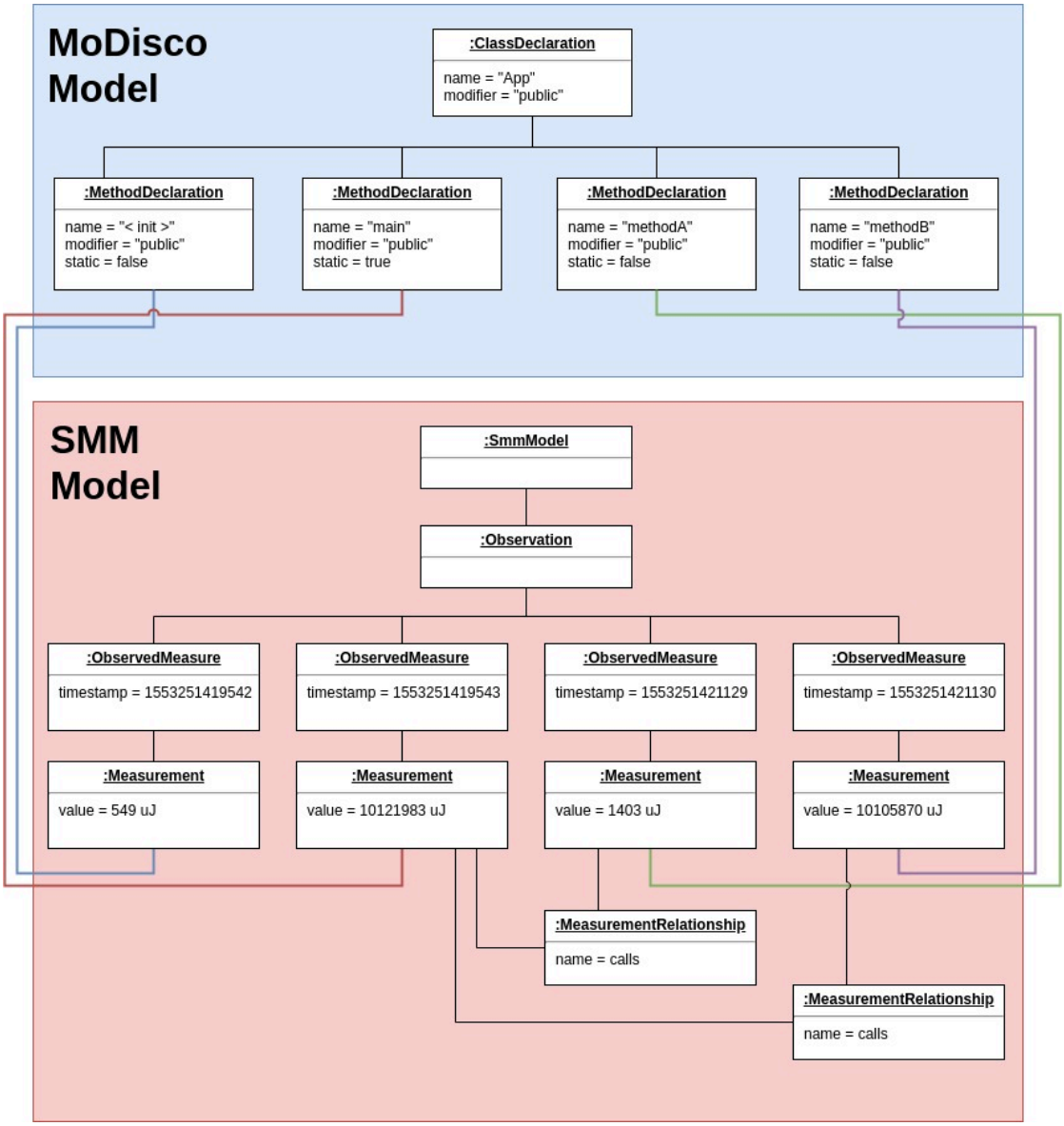
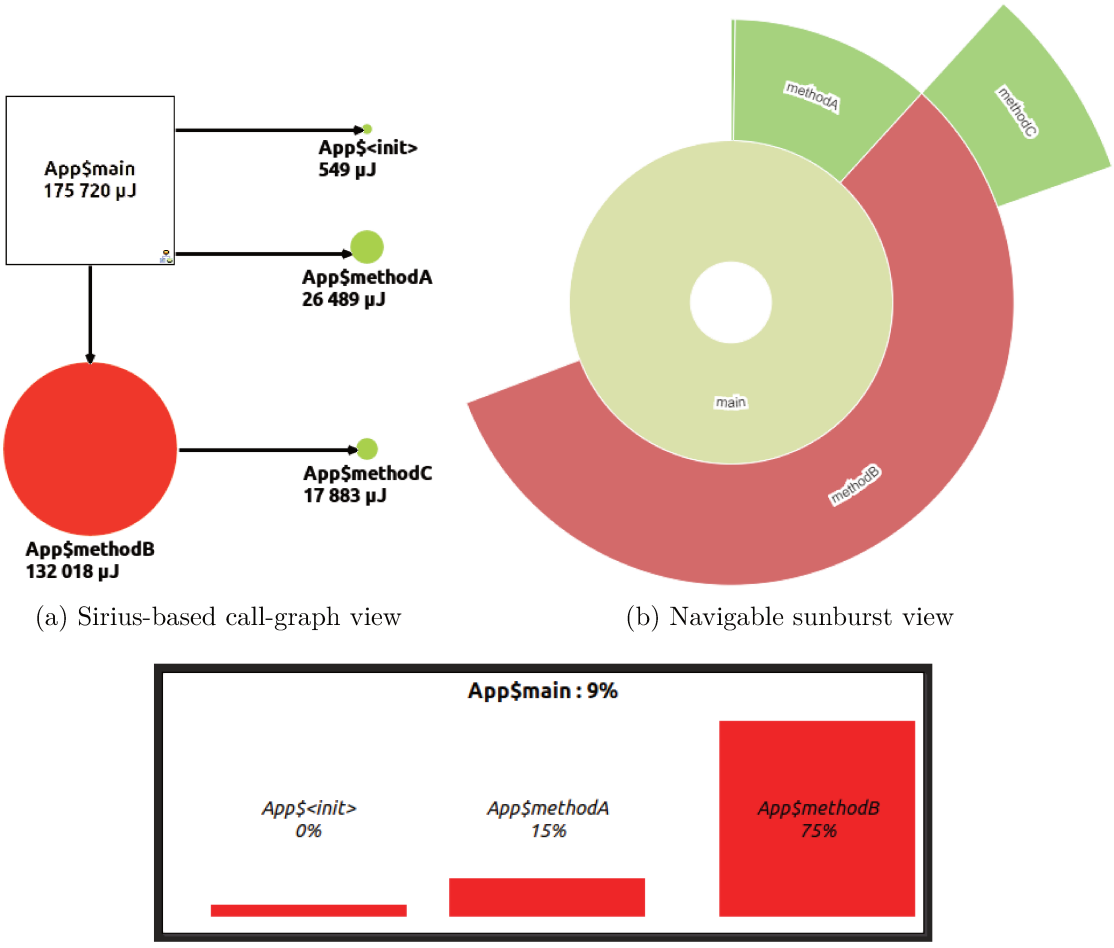


Figure 3.9: Excerpt of the source code model (the `MethodDeclaration` instances) characterized with energy measurements

Sirius graphical modeling workbench that display the program's call graph and its energy consumption, in order to help developers locating energy consuming methods. We also propose a more compact view implemented using the sunburst radial visualization system [182]. This last view is not defined using model-driven techniques, at opposed to the Sirius-based views, and is thus less maintainable, but the library used provides high-quality features for navigation. We display these



(a) Sirius-based call-graph view

(b) Navigable sunburst view

(c) Sirius-based single method view

Figure 3.10: Three views to display software energy consumption.

three views in Figure 3.10.

Second, the program can be refactored and optimized by only working at the model level. Programmers can specify transformation rules for refactoring [68], without having to go back to the source code level, in order to optimize the energy consumption using existing energy-efficient practices [149, 158, 171, 179]. As stated by G. Pinto et al., refactoring approaches focusing on improving the energy consumption of programs are lacking [154]. Our model could help programmers refactoring their applications towards that purpose, by specifying a model transformation to apply to the source-code model, and generating the program optimized. The energy measurements of the model can be used as predicates for the transformation. For instance, a model transformation could be specified, only

targeting the methods that consume more than a specified energy. As an example, it could transform all the collections used to more efficient implementations, or use less consuming data types (e.g., floats instead of doubles), as it has been shown that it can reduce the energy consumption [116]. The modified source code of the program can then be generated again, using MoDisco code generator, and be used to consume less energy. More complex usages could be imagined, such as performing random model transformations on energy inefficient features and comparing the energy consumption of the regenerated code, until a smaller consumption is found, as proposed in design space exploration refactoring approaches.

3.3.6 Threat to validity

Several points have to be considered, in regard to the validity of our approach. First of all, our current model does not aim at containing the precise energy consumption measurement for each method. It stores in a standard model the information we gather from our measuring tools. In our implementation we used RAPL to measure the energy consumed by the program. RAPL induces an overhead, that has not been quantified here, and which threatens the accuracy of the measurements. Furthermore, RAPL also includes the energy that the operating system is consuming. For that reason, the energy measurements we are able to gather with our current tooling might not be accurate. Using a dedicated system for energy measurement could be used to estimate this difference.

3.3.7 Conclusion

In this section we re-use the static source code model in which we injected execution traces as presented in Section 3.2. Thus, we propose an approach for modeling the source code of an application and decorating it with energy measurements. The instrumentation step described in Section 3.2 is extended to also perform energy measurements in addition to tracing the execution of the software. The measurements are dynamically gathered in the SMM model, and associated with the source code model. This model can be used by programmers as a basis for analyzing and optimizing programs, and offers a better energy-awareness, useful for implementing greener applications. This shows one usage of our dynamic model for improving sustainability. In the next section, we rely on this dynamic model to perform impact analysis on changes in order to reduce the cost of the testing phase, an other usage fostering sustainability with models.

3.4 Trace model applied to regression test selection

3.4.1 Introduction

In this section, we propose a different usage of our software model enhanced with execution traces, presented in Section 3.2. Instead of modeling energy-related information, we use this model as a dependency graph, to perform impact analysis of the source code changes. This impact analysis is then used to lighten the cost of regression testing.

Regression testing (RT) is an important step in the software development lifecycle. It ensures that code updates do not break the functionalities that have already been successfully tested. However, the size of regression test suites tends to grow fast [205] when an application is evolving; thus considerably increasing the testing cost: both in *time* and *energy* consumptions.

According to various studies, up to 80 % of testing cost is related to regression testing, and more than 50 % of software maintenance cost is dedicated to testing [61]. The process of source code compilation, load, and test execution is commonly called a *build*. When a build is over, a result, successful or not, is returned to the developer. According to this result, the developer will either go to another task or correct the code that has regressed. This is especially true in the context of Continuous Integration (CI), where the regression testing takes place on a separate server [74]. During the time of the build, the developer does not know yet if she needs to correct the code (the wait could be long: e.g., 25:33 min including 06:04+18:11 min testing time for the build `372209560` of the Google Guava project[5]). Thus, reducing this duration would improve the development productivity.

Reducing the cost of regression testing by only running a specific subset of test cases is the purpose of Regression Test Selection (RTS). Most of the large variety of RTS techniques are based on change impact analysis [106, 109]. When an application under development is being modified, it might be unnecessary to run all the test cases, especially the ones that are not impacted by changes in the source code. For instance, existing RTS techniques such as Ekstazi [70] are able to reduce the regression test time for the Google Guava project [6] to an average of 45 %. A standard RTS approach usually involves three phases:

- (C) Collection of the dependencies between code and test cases.
- (A) Analysis of the changes to select impacted test cases.

[5] https://travis-ci.org/google/guava/jobs/372209560
[6] https://github.com/google/guava

- (E) Execution of the selected test cases.

The benefit of running less test cases during phase (E) could be counterbalanced by the overhead introduced by the phases (C) and (A). Some existing approaches such as FaultTracer [209] have shown a RTS time that is longer than the execution of all tests. For that reason, several RTS approaches named *offline* calculate the phase (C) beforehand. Hence, when the user starts a build, the dependencies are already available for performing phases (A) and (E) with a positive time gain [208].

Figure 3.11 shows a sequence diagram of an offline RTS approach. The first build corresponding to revision R_i, is performed normally, and all test cases are executed. Then, dependencies are collected (i.e., phase C) using this revision, *offline*. Later, when the developer proposes a new revision R_{1+1}, the dependencies previously computed are used to locate the test cases impacted by the changes (i.e., phase A). The reduced test suite is executed (i.e., phase A). Finally the developer is notified, and these three phases can be reproduced at each revision.

The overhead introduced by the phase (A) depends on how the dependencies are computed during the phase (A) and analysed during the phase (C). It mainly depends on the *precision*. A RTS technique is said to be *precise* if all selected test cases are affected by the changed code [42] (i.e., no useless test case is ran on unchanged code). Being *precise* is not mandatory and computing the dependencies at a really fine grain induces a significant overhead [70]. The balance between the precision and its induced overhead should be considered when developing and using RTS techniques.

The precision of RTS techniques depends on the granularity when considering *source-code updates*. It could distinguish *file*, *class*, *method*, or *statement* updates. For instance, a class-level update summarizes all the modifications inside a class whereas a statement-level update considers modifications of a single statement. The usage of statement-level updates is more precise but induces more overhead than file-level updates. This is the reason why existing RTS approaches [70] have shown faster end-to-end[7] results using file-level changes, despite selecting more tests. Nevertheless, Lingming Zhang shows that depending on the file updated and the case-study, it could be worth considering finer method granularity [208]. Therefore, by designing a model-driven RTS approach, we can provide **modularity** and allow the tester to manage the granularity and optimise the precision. Furthermore, the overhead induced by a precise approach can be mitigated by performing the dependency collection phase (C) while offline.

Independently from the precision, RTS techniques must be *safe*, meaning that they should select every test case that is impacted by changes in the code [168].

[7] time necessary to compute the set of test cases to run, plus the test execution time.

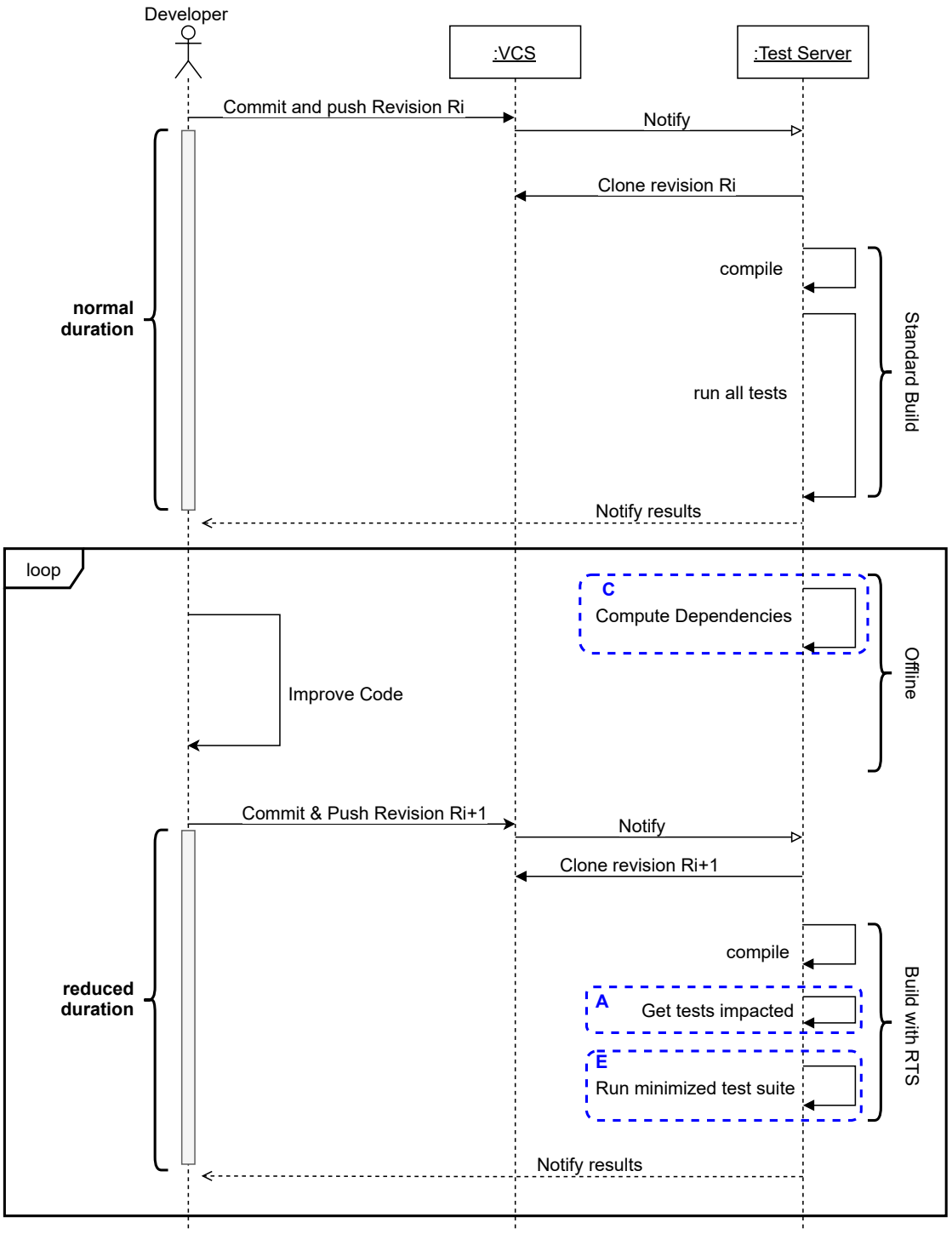


Figure 3.11: Sequence diagram of an offline RTS.

While most of the related work on RTS is dedicated to ensuring precision or performance of language-specific and RTS dedicated tools, in this thesis we focus on providing a solution that leverages MDE for improving RTS in terms of its **reusability** and **interoperability**. The accuracy of model-driven program analysis techniques has proven this approach to be suitable for tracing down the execution of programs, as shown in Section 3.2, and can benefit to RTS. First, the modularity of MDE allows for a configurable approach where the RTS precision can be set as required. Second, regression testing is one step among all the software development steps, MDE aims to prevent each step to be independent by sharing the information via models. Therefore, we propose a model-driven RTS approach that relies on the execution traces available in the *impact analysis model*. That model is used all along the RTS and then it can be exchanged, completed, and reused for different purposes (e.g., debugging, performance analysis, optimization). Such model can be connected to the other models of the MDE environment, allowing for instance to trace regressions from the requirements model or UML models, etc. Finally, regressions often happen because of changes in different artifacts around the code, such as resources, configuration files, or data files. The holistic view fostered by MDE would enable to address all these problems in a uniform way based on the impact analysis model.

The risk of applying MDE in RTS is its potential performance overhead, in particular for building the model [176]. In this section we show that this cost can be reduced by developing an *offline* model-driven RTS technique: costly impact analysis model creation is anticipated. Our resulting tool significantly reduces the execution time of regression testing in our case studies, by up to 32 %. While state-of-the-art dedicated RTS tools may have better performance, we argue that the reusability of our computed models makes our solution especially valuable in MDE environments.

In this section, we propose a model-driven approach for a highly *precise* and *safe* RTS using statement-level impact analysis, in order to select test methods impacted by code changes. We build a dynamic model of execution traces offline, using the approach presented in Section 3.2, and leverage it for impact analysis. When a user requests a build, this model is queried to select all the tests impacted by changes. Finally, this reduced set of test cases is executed, thus accelerating the regression testing time. The approach is based on the customizable model storing the execution traces of the tests. If the fine-grain accuracy of this model may slow down the RTS, it is a powerful asset for other software engineering tasks, hence improving its reusability.

To evaluate this approach we aim at answering to the following questions in this section:

- **RQ1**: Does our model-driven RTS approach allow for a *safe* and *precise*

RTS?

- **RQ2**: Is our model-driven RTS approach efficient enough to significantly reduce the regression testing time?
- **RQ3**: Does our model-driven RTS approach offers better modularity and reusability than existing approaches?

The remainder of this section is organized as follows: Section 3.4.2 describes a motivation example, then Section 3.4.3 introduces our approach, followed by an evaluation driven by several experiments in Section 3.4.4. Finally, Section 3.4.5 discusses our proposal and Section 3.4.7 concludes this section.

3.4.2 Running Example

Figure 3.12 presents an example of RTS, and is used to define the context of our model-driven approach for RTS. Figure 3.12a and Figure 3.12b define a first and a second revisions of a Java program, respectively. They are verified using the JUnit testing framework.

In the first revision, R_1, the test class `TestCA` verifies that the class `CA` works as expected. Four test cases are implemented: `test_mA_0`, `test_mA_1`, `test_mB`, and `test_mC`. The first two tests cover the two branches of the *if* condition in `mA`. The third one tests the method `mB`, and the last one tests the method `mC`.

The source code modifications of revision R_2 update several lines. The first change occurs at line 4, where the statement inside the *if* branch is updated. The second change occurs at line 8, by renaming method `mB` to `mD`. Finally, the last modification considers the new method name `mD` inside the corresponding test case.

Orso et al. define two different kinds of program changes, *statement-level* changes and *declaration-level* changes [141]. A *statement-level* change can be either a modification, deletion, or addition of executable statements, such as lines 4 and 27 changes. Note that statement-level *impact analysis* relates to the computation of the impacts of a statement modification, whereas a statement-level *change is* a statement modification.

A *declaration-level* change is a modification of a signature, such as line 8 change. It could be method name modifications, method additions or deletions, variable type changes, or any kind of signature changes.

Depending on the granularity of the change impact analysis, different test sets might be selected for a re-run, as illustrated in Table 3.1. Applying RTS with Class granularity selects four test cases, with Method granularity three, and with Statement granularity only two, thus improving the precision. For instance, a Method granularity would consider the entire method `mA()` changed, and would select `test_mA_1()` even if it is not impacted. To improve the precision, Statement

```
public class CA {
  void mA(int i) {
    if (i == 0)
      doSomething1();
    else
      doSomething2();
  }
  void mB() {
    doSomething3();
  }
  void mC() {
    doSomething4();
  }
}
public class TestCA {
  @Test
  void test_mA_0() {
    new CA().mA(0);
  }
  @Test
  void test_mA_1() {
    new CA().mA(1);
  }
  @Test
  void test_mB() {
    new CA().mB();}
  }
  @Test
  void test_mC() {
    new CA().mC();
  }
}
```

(a) Revision R_1

```
public class CA {
  void mA}(int i) {
  if (i == 0)
    doSomething5();
  else
    doSomething2();
  }
  void mD() {
    doSomething3();
  }
  void mC() {
    doSomething4();
  }
}
public class TestCA {
  @Test
  void test_mA_0() {
    new CA().mA(0);
  }
  @Test
  void test_mA_1() {
    new CA().mA(1);
  }
  @Test
  void test_mB() {
    new CA().mD();
  }
  @Test
  void test_mC() {
    new CA().mC();
  }
}
```

(b) Revision R_2

Figure 3.12: Two revisions of a program

	Granularity		
Methods	**Class**	**Method**	**Statement**
`test_mA_0`	x	x	x
`test_mA_1`	x	x	
`test_mB`	x	x	x
`test_mC`	x		

Table 3.1: Selection of test methods depending on the granularity of source-code updates

granularity analyses the impact of each statement: `test_mA_1()` is not impacted since it does not run line 4. Therefore, the test execution time is reduced but the overhead is increased.

In the next section, we present a model-driven approach that is able to select impacted test cases at the most precise Statement granularity. Thereafter, since our approach provides a persistent impact analysis model, it will be up to the tester to choose the granularity to balance overhead/precision w.r.t. her case study.

3.4.3 Approach

Our model-driven RTS approach performs the three RTS phases relying on the impact analysis model built as shown in Section 3.2:

- (C) Computation of the impact analysis model.
- (A) Analysis of the impact analysis model to select the impacted tests.
- (E) Execution of the impacted tests.

Computation of the impact analysis model

Building the impact analysis model can be performed exactly as shown in Section 3.2. Using the Figure 3.12a as an input program, the impact analysis model produced by our approach is the one displayed in Figure 3.13. For readability purposes, we simply display the relationship between statements and test cases a "*trace*", instead of the SMM measurements and relationships presented earlier. This impact analysis model is performed offline, and persisted in order to be used as soon as a new version of the code needs to be analyzed, to immediately select the test cases impacted. Building the model corresponds to the *collection* phase (C) of the RTS.

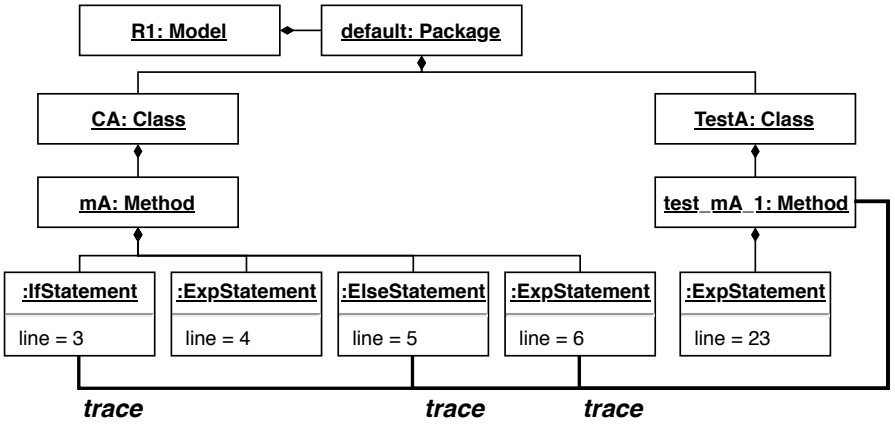


Figure 3.13: Excerpt of the impact analysis model

Using the model to select impacted tests

The second RTS phase selects the test cases to run, based on the impact analysis model in our model-driven approach (E). This phase is online, starting when the user launches the regression test activity, e.g., after pushing a new version of the code on a Version Control System (VCS).

Once the build starts and before anything else, the new revision is compiled. If the compilation fails, no tests are executed and an error report is given to the developer. If the code compiles successfully, then the RTS can be performed. The two revisions of the source code are compared, in order to locate the changes. It is done with a VCS comparison algorithm (we use the one implemented in JGit[8]).

Running such algorithm on two different revisions of the code produces a set of *DiffEntries*. They are not only produced from Java files, but from any kind of files contained in the project, such as configuration files, and compiled sources. When a *DiffEntry* concerns a Java file, the impact of source code changes on test cases is computed using the Impact Analysis Model. We do not consider non-Java files yet, however existing work from the state of the art proposed an approach to do so: the instrumentation has to specifically observe libraries that read and write files (i.e., `FileInputStream` and `FileReader` for Java), and add the files to the dependency model [70].

The changed statements are gathered in the model and the impacted test cases are selected, using the *trace* reference. For instance, if we focus on the Figure 3.13, revision R_2 in Figure 3.12b modified the `ExpStatement` line 6. Using the *trace* reference in the model, `test_mA_1` is considered as impacted, and is selected to be run.

[8] https://www.eclipse.org/jgit/

As shown in Figure 3.13, the *trace* references are precise, linking statements and test methods. Moreover, by querying the model we can derive the impact at coarser levels of granularity. For instance in Figure 3.13, the `ExpStatement` line 6 is contained inside the method `mA`, itself contained inside the class `CA`. Hence, a coarser-grained analysis (i.e., method or class granularity) can determine that modifying the method `mA` or the class `CA` would impact `test_mA_1`, and so on. Implementing a faster hybrid visitor for computing a coarser-grained impact analysis, such as the one proposed by Lingming Zhang [208] is possible, thanks to the reusability of the model. Indeed, it is possible to: (1) Build a coarser model during the collection phase (C), by only considering the dependencies at a coarser level (method, classes), this would accelerate the offline model building phase, or; (2) Compute the impacts at a coarser level during the impact analysis phase (A), this would select more tests, but would accelerate the impact analysis phase.

Each *DiffEntry* considers a block of a few lines of text, and is classified according to the type of update (Modification, Insertion, Deletion). In the following we detail the behavior of the test selection by type of update.

Modification

Modifications introduced by developer can be either statement-level changes or declaration-level changes. If the update is a statement-level change, then the impacts can be computed with a statement-granularity. If it is a declaration-level change, then the impacts are computed at a coarser granularity (Method, or Class), depending on the change.

Those changes are either in the SUT, or in the test cases. When a test case is modified, it is immediately selected to be re-executed. However, the approach is more specific for SUT modifications: In case of statement-level changes, the modified lines are parsed, in order to get a set of modified statements. Those statements are queried in the model, and using the *trace* reference, all the test cases having executed this specific statement are selected for a new run.

The approach differs in case of declaration-level changes: If a method declaration is modified, by either renaming it, or changing the signature, this method is queried in the model, using its old-version signature. Then, all the test cases that executed this method are gathered, using the *trace* reference, and selected for a re-run.

Furthermore, the same reasoning is applied with declaration-level changes at class level. Thus, when a class signature is modified, the old version of the class is queried in the model, in order to get the test cases impacted, using the *trace* relation.

Selecting the tests impacted by the modification at different granularities (e.g., Package) is also possible, by applying the same approach on the model.

Insertion

When the change is an insertion, several cases are considered:

(1) Any test-related insertion results in the selection of the test case. Those can be either a new test, or an existing test with new lines of code.

(2) When new lines are added in an existing SUT method (i.e., statement-level changes), the modified method is fetched inside the Impact Analysis Model, using an OCL query. And thus, impacted test cases are obtained using the *impacts* reference between this method and the test cases.

(3) When new classes or methods are added: if the method was not present in the previous revision, then it is not present in the model either, and hence, no impacts can be directly computed from it. However, adding such method can have impacts on the existing SUT. Figure 3.14 describes a situation where the insertion of a new method has impacts on the test cases. This case can be solved by checking by OCL if such method overrides a method of a superclass. For instance, if a method $m()$ is added in class C, this approach would check if $m()$ exists in superclass $SuperC$. If it does, then all the test cases executing `C.m()` would be selected for an execution, using the *trace* reference of `SuperC.m()` in the Impact Analysis Model. This scenario adds a minor overhead, but is necessary to ensure the safety of our RTS approach.

Deletion

In the same way as in the other changes, deletions can be at the *statement-level*, or *declaration-level*, which implies three different behaviors:

(1) As a statement-level change, when a line of code is removed at the statement level (i.e.,), inside a method, this method is fetched by querying the Impact Analysis Model. Finally, its impacts on the test cases are obtained using the *trace* reference. Test cases impacted are then selected to be run again.

(2) As a declaration-level change, when an entire method, which does not override any method from a parent class, is removed, then the test cases calling it have probably been modified too. Indeed, removing a non-overridden method from a class, but not the calls to this method would produce compilation issues. The same reasoning works at the class-level. If a deleted parent class is used in the test cases, then those tests have to be updated too.

(3) If the deleted method overrides a method from a superclass, then the old version of this method is fetched from the model, and all the tests executing it are selected, using the *trace* references.

To conclude the Section 3.4.3, we notice that we successfully design and implement a model-driven RTS approach that can select impacted test cases based on a

```
abstract class SuperC {
  void m(){
    doSomething();
  }
}
class C extends SuperC {




}
class TestC {
  @Test
  void test(){
    new C().m();
  }
}
```

(a) Revision R_1

```
abstract class SuperC\{
  void m() {
    doSomething();
  }
}
class C extends SuperC {
  void m() {
    doSomething2();
  }
}
class TestC {
  @Test
  void test() {
    new C().m();
  }
}
```

(b) Revision R_2

Figure 3.14: Adding a new method impacts an existing test case

model. The current version of our prototype is available on GitHub[9]. Thanks to that model, the granularity of the selection could be easily adapted according to the needs. As shown in Section 3.4.3, it is possible to get the impacted test cases at several granularities (i.e., statement, method, class), depending on the change type (statement-level, declaration-level). Thus, in order to accelerate the RTS, it is possible to limit the impact analysis to a certain granularity.

3.4.4 Evaluation

This section presents an experimental evaluation of our approach. First, we present the environment setup and then the experimental workflow, followed by a presentation of our results.

Setup

All the experiments are executed on an Intel Core i5-7200U CPU (2.50GHz), 8GB of RAM, running with Ubuntu-16.04, and using Java 1.8.0_51. The four projects

[9] https://github.com/atlanmod/MDE4RTS

Framework	First commit	LOC	#Classes Tested	#Test Classes
Javalin	caae71e	2500	13	23
JSoup	7f8010d	25 000	61	31
JUnit4	64155f8	100 000	126	174
AssertJ	cf4d367	250 000	368	1903

Table 3.2: Projects used for evaluation

presented in Table 3.2 are used for our evaluation. The column *First commit* corresponds to the first Git revision used, *LOC* approximates the lines of Java code available in each project, and the last two columns list the number of system under test classes and number of test classes, during the first commit. Those projects are available on GitHub[10] [11] [12] [13].

These four frameworks have been chosen because they answer to several criteria that were mandatory for the current state of our prototype: using the git VCS, Java, the Maven dependency manager, and having test suites running with the JUnit testing framework. Since all these tools are popular in the development community, restricting the usage of our prototype to them does not significantly hamper its practical usefulness. Furthermore, the execution trace modelled with these frameworks are small enough to fit within the RAM allocated to the JVM. Finally, these frameworks contain both declaration-level and statement-level changes.

Workflow

We used the following workflow to run the experiments. First, a specific starting revision is determined. This can be a previous tag in a Git history, for instance. This revision is called R_1. Then we define a variable *Step*, representing the number of commits between each computation of the RTS. For instance, with $Step = 2$, the Regression Test Selection would be computed between R_1 and R_3. Instead of applying the RTS at each commit, being able to merge multiple commits at once offers a more accurate representation of a development lifecycle. In practice, developers either push multiple commits at once, or use pull requests, to merge several commits from other branches on the VCS. All the projects are not developed following pull-based workflow [74], we use the *step* variable instead.

We then define another variable, Max representing the maximum amount of commits to analyse. For instance, with $Max = 50$, the last revision that could be analysed would be R_{50}.

[10] https://github.com/jhy/jsoup
[11] https://github.com/joel-costigliola/assertj-core
[12] https://github.com/junit-team/junit4
[13] https://github.com/tipsy/javalin

The experimentation starts with the revision R_1. Since no model can be found, all tests are executed. Then the impact analysis model is built offline, thus computing the dependencies at the statement-level. Once this is done, the revision R_{1+Step} is cloned, and the RTS is applied between R_1 and R_{1+Step}, using the impact analysis model previously generated. Hence, the subset of tests is executed. The impact analysis model is built, from the revision R_{1+Step}, and the next revision $R_{1+2Step}$ is cloned, and this goes on until reaching R_{Max}. Each time, both the subset of tests and all tests are executed with the same technique, in order to compare the results.

For the experimentation, the Max value is set to 100, and $Step$ is set to 5. In their work, Georgios Gousios et al. evaluate the average number of commits per pull request to be 4.47, and with 90% of pull requests bundling 6 commits or less [74]. A $Step$ value of 5 is in that range. Finally we consider a Max value of 100 to encounter every type of change.

Results

Figure 3.15 presents the results of applying our model-driven approach to RTS to a pool of 100 commits for each framework evaluated. The first section of the table presents the quantity of test methods selected by our approach. As the number of tests varies along the commits analysed, average values are used. The second section reports the average compilation and execution times for each project, without RTS. Hence, summing these two values for each project shows the standard build time. The third section presents the average times of the three model-driven RTS activities. The sum of the compilation time with the identification and execution of impacted tests results in the duration of a full build, when using our approach. This model-driven RTS build durations and the standard build durations are compared in the last section of the table, as well as in Figure 3.16.

Results show that applying model-driven RTS shortens the build time, in all cases, when the impact analysis model is built beforehand (i.e., offline. This duration corresponds to line (C) in Figure 3.15). On a bigger project such as AssertJ, the results are more significant, with a RTS build time lasting 67.5 % of a standard build, when only 18.24 % of the tests cases are executed. However, for a project of small size, like Javalin, the impact on the build time is less significant, since selecting and executing the tests takes 90.3 % of the execution time of all tests.

As explained earlier, running the tests, either all tests, or a test selection, includes a compilation time. This is performed using Maven and hence several steps of the Maven build lifecycle have to be executed: validation, compilation, and finally testing. Comparing only the actual test execution times would show

Frameworks	**JSoup**	**AssertJ**	**JUnit4**	**Javalin**
avg Selected test methods	200	888	35	6
avg Test methods	588	4868	998	95
% **Test methods selected**	34.01%	18.24%	3.51%	6.31%
(1) Compilation Time (ms)	4805	9854	10 230	18 368
(2) Executing all tests	8051	34 762	11 828	4893
(*C*) Computing impact analysis model (ms)	78 307	375 747	82 478	50 555
(*A*) Identifying impacted tests (ms)	649	4464	1081	331
(*E*) Executing impacted tests (ms)	4079	15 791	3549	2635
Standard build: (1) + (2)	12 856	44 616	22 058	23 261
Model-driven RTS build: (1) + (A) + (E)	9533	30 109	14 860	21 003
((1) + (A) + (E)) / ((1) + (2))	74.15%	67.5%	67.4%	90.3%

Figure 3.15: Average evaluation results

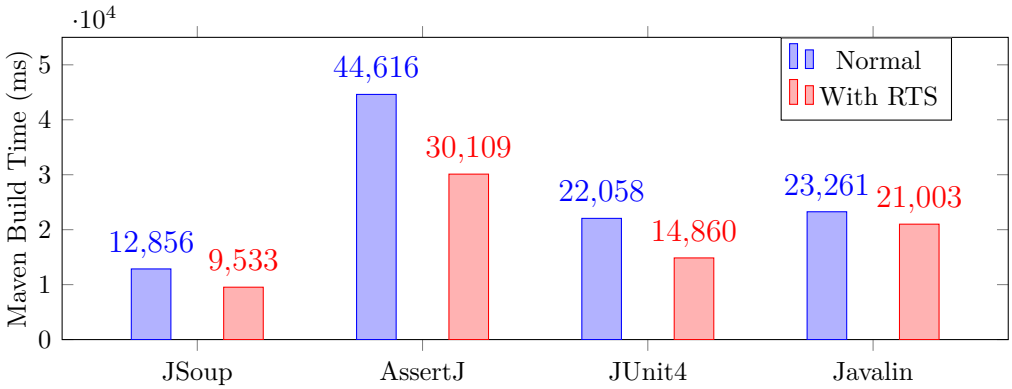


Figure 3.16: Build times with and without RTS.

a bigger gap between RTS and the execution of all tests, but would not reflect real-life builds.

In our experimentation, selecting the tests cases and running them takes always less time than running all the tests cases. As predicted, a loss of time may occur when the number of tests is low since the run of a test subset will not compensate the overhead of the RTS phase (A). However, testers will consider RTS only when their test suite length starts to be problematic and in that case, our approach is

beneficial.

3.4.5 Discussion

In this section, we discuss the approach and the experiments to answer the three questions asked in Section 3.4.1.

RQ1: Precision

Precision has been discussed previously in Section 3.4.3. We can answer the second part of **RQ1** positively since our model-driven approach allows precise RTS. Thanks to the impact analysis model, the precision can be as much finer as Statement granularity. Moreover, the modularity of the approach allows to adapt the precision by analysing differently the impact analysis model.

RQ1: Safety

To answer the first part of **RQ1**, we consider the *safety* of our model-driven RTS approach. We should prevent an impacted test not to be selected. Therefore, we consider four risks: miss a source code change, analyse a non up-to-date impact analysis model, consider all the software artefacts, and flaky tests.

All the tests are first executed to generate the impact analysis model and their execution is traced at the finest statement granularity. Hence, if any statement is modified in a future revision, then all the tests that previously executed it can be selected thanks to the *trace* references.

The test selection requires an impact analysis model, built offline. If the user launches regression testing too frequently, the model of the last revision may not have been completely built. If it happens, the last entirely computed model is used instead. It would be less *precise*: since more code changes would be considered, hence more test cases selected, but the *safety* is ensured. If no model is available, then the entire test suite is executed.

The current state of our tool does not compute the impact of external files changes. Nonetheless, such dependencies could be added in the impact analysis model: e.g., Gregoric et al. describe an efficient approach to collect file dependencies, using BCI and monitoring [70]. All standard library methods able to access files are monitored, and dependencies towards these files are added when accessed at runtime. Enhancing our current SCI to monitor external files accesses is an interesting perspective to add to our model-driven approach for RTS, and perfectly feasible as the static MoDisco model we rely on includes external files too. We ensure the safety in the current implementation by detecting external files modification

(using a `diff` algorithm), and warning the developer. She can then run all the regression test cases to ensure that nothing has been broken.

Finally, non-deterministic tests, also known as *Flaky Tests* [118], are not covered by our approach, nor by state-of-the-art RTS approaches. These tests can behave differently between several executions, (depending on multiple parameters: asynchronous calls, concurrency, network, test order dependencies, etc.) and therefore cannot be traced deterministically. The safety risk of Flaky Tests is not only affected RTS but all the testing process and it is up to the tester to annotate them for being processed carefully (e.g., with several execution).

RQ2: Performance

As presented in Section 3.4.1 with **RQ2**, our model-driven approach has to be efficient enough to reduce the regression testing time. Poorly performed RTS steps might produce an overhead so important that selecting and executing a reduced set of tests would take longer than running the entire test suite.

First, our RTS approach is offline. Therefore, we do not take into account the model generation time (RTS phase (C)). This phase is costly when generating an impact analysis model, but it has already been performed before the tester launches the regression testing. Second, the test cases selection (RTS phase (A)) is critical, and must be computed in a fast way to prevent a counter-productive overhead. Two expensive activities are processed during the test cases selection:

1. Comparing two revisions of a project, more specifically at a fine granularity (Statement-level).

2. Querying the model for specific elements.

Being efficient on these two points is a major concern. Performance when manipulating models is a known issue in model-driven engineering. Experiments presented in Section 3.4.4 show that we positively answer **RQ2** since selecting the tests cases and running them takes always less time than running all the tests cases. In addition, our model-driven approach allows to configure the granularity and then to balance the overhead depending on the case study (e.g., a case study with few test cases would benefit from a coarser-granularity, while a case study with long-running test cases would benefit from a fine-granularity to reject slow and non necessary test cases).

Our performance (gain of up to 32 %) are less important than the ones of dedicated tools such as the ones of [208][70] (they can gain more than 40 %). However, as we will see, our generic model-driven approachenables several kinds of analysis, like the one shown in Section 3.3.

RQ3: Complementary use of the Model

In a MDE environment, our impact analysis model becomes a part of the holistic modeling of all software aspects. Thus, it can be reused to perform heterogeneous analysis on the project. For example, tracing down the execution of test cases inside a model would provide a better understanding of a test suite, useful while reverse-engineering old code bases. This model could also be used to get an accurate overview of the testing code coverage, and thus significantly improve the quality of test suites [127].

Concretely, in our tool, we use the Structured Metrics Meta-model[14] (SMM) for gathering measurements performed during the test execution, as shown in Section 3.3.2. SMM enables the modelling of several metrics: lines of code, test execution durations, method complexities, energy consumptions, and so on. By weaving the SMM model along with the model produced during RTS, we are able to know the energy saved by not running the test cases that have not been impacted. Thus we successfully answer the **RQ3** since the impact analysis model can be completed and analysed for other software engineering analysis. On the contrary of dedicated tools that are not compatible and require each one to run and to analyse the program, we collect and serialize all the information in one impact analysis model. Created once, we save time, counter-balancing the overhead induced by model manipulations.

3.4.6 Threats to the validity

Several threats to the validity of our experimentation have to be considered:

Commits analyzed

First of all, the amount of selected tests highly depends of the changes made by developers. For instance, the modification added between commits `fce43e6` and `4fe13cb` on the project AssertJ impacted 3312 existing test methods. Therefore, the results could be radically different using other commits. Furthermore, two projects developed by programmers following different ways of committing code update will result in different number of changes between revisions [6].

Nonetheless, by applying our RTS approach on a set of 100 successive commits, we assume that it covers usual development scenarios, such as adding of new methods, or modifying of the core methods of the project.

[14] https://www.omg.org/spec/SMM/

Single machine

All those results have been computed on a single computer dedicated to this task. Even if a particular attention has been taken to only dedicate the machine to the experimentations, background operations performed on the system may have impacted the execution times reported.

State of the prototype

Our research prototype's implementation may contain undetected bugs, and thus the results produced may be subject to variations with the upcoming improvements and corrections. Even if this prototype is well tested and its source code has been thoroughly examined (and is open to inspection), not all the results obtained during the experimentation can be manually verified. Especially the safety of the approach, while theoretically valid, has been manually checked only for the small projects. Finally, this prototype is still under development, and so far its performances have been improving at each revision.

Scalability

Models are well known for suffering of scalability issues [143]. In fact, the standard EMF implementation persists models using XML data structure. Those files cannot be partially loaded, and if the model is too large, it might not fit in memory anymore. When the source code is massive (thousands of classes), the model generated by MoDisco can be big (GBs), which may produce scalability issues, especially when querying elements. This limitation in space can be resolved by using scalable persistence layers for models, such as NeoEMF[15], or CDO[16]. However, it does not solve execution time issues.

3.4.7 Conclusion

In this section of the thesis, we presented an approach that fosters reusability of our model of execution traces in the context of RTS. We use the trace model, presented in Section 3.2, starting from a source-code structural model and enhanced with dynamic information, gathered during test case execution. On top of this model we have implemented a fine-grained impact analysis, allowing for effective RTS. Computing the long-running impact analysis offline allows us to rapidly apply the RTS when the user needs it, minimizing the waiting time, and hence improving the productivity. Indeed, experiments on four different open-source projects showed that

[15] https://www.neoemf.com/
[16] https://www.eclipse.org/cdo/

our approach significantly shortens the regression test execution time. Moreover, the proposed impact analysis model can serve as a basis for even more software engineering activities: fault localization, precise code coverage, etc.

While the current version of our prototype provides positive performance results, several points can still be improved. Incrementality should be considered to reuse and complete the model during the RTS phase. Indeed, after each build, our prototype creates the whole model from scratch, using MoDisco. Using *diffs* to update an existing model according to the new source code version would shorten the impact analysis model building time.

3.5 Chapter conclusion

This chapter presented three contributions leveraging MDE for modeling execution traces of Java applications. We capture the execution of programs using instrumentation techniques, and inject into a source code model the traces. The reusability of MDE enables many usages of these traces, in different domains.

We present an approach for energy modeling: the energy consumption of the program's artifacts is measured dynamically, and modeled along with the trace, providing a bird's eye view on the energy consumption of its application to the developer. This representation can then serve as a basis for energy-aware model transformations and refactoring.

A second usage of these traces is proposed through RTS. The trace model is used as a dependency model for impact analysis on source code changes. This enables a faster regression testing phase.

However, if these usages enable a better sustainability in standard software engineering, applying them to the domain of CPS, where software can be distributed over complex networks, is more complex. In the next chapter, we propose an approach to model and trace the execution of CPS.

Chapter 4

Model-driven monitoring of CPS

4.1 Introduction

As mentioned in Chapter 2, a CPS refers to a system that is composed of both physical devices and software artifacts. Monitoring a CPS is essential to understand it, control its behavior, and perform any kind of dynamic analysis on it. Over the past years, the number of CPS increased quickly, and is expected to grow even quicker in the forthcoming years [99]. This would make the design of their monitoring applications more and more challenging, as their complexity gets more important. In this section, we consider this problem of designing CPS monitoring application, and study it in the context of a specific type of CPS: Sensors and Actuators Network (SAN). Sensors are small entities mainly used for sensing, processing and/or sending data, whereas Actuators are controllable entities requiring external input [5, 145]. Such network requires low latency, valid data output, good coordination between sensors and actuators, and a long lifetime. To fulfill these requirements, being able to efficiently monitor the SAN is mandatory.

A common way of designing the monitoring of a CPS is to setup physical devices to be monitored, and configure and adapt existing tools to combine them into a monitoring application dedicated to this CPS [54]. It requires a dispensable and error-prone dependency: the implementation of the physical part of the CPS needs to be available first, in order to drive the implementation of the monitor's software part. This makes such monitoring application complicated to develop, maintain and update.

Applying MDE to SAN has been investigated [8, 157, 167], and languages such as SensorML [24] based on a meta-model, or Sosa [92] and Ssn [134], based on ontologies, have been proposed. However, to our knowledge, no MDE approach to monitor the SAN execution while analyzing and persisting measurements has been proposed. Even if a standard *Structured Metrics Meta-model* (SMM) [138] exists, as described in Section 2.1.4, it is not integrated into SAN modeling languages. This results in a lack of modularity and reusability since maintenance of both parts (the SAN itself and its monitoring applications) is made separately.

Therefore, in this chapter we leverage MDE for CPS traceability and monitoring, and propose the three-step approach illustrated in Figure 4.1. First, the SAN is designed using a model, conforming to a SAN meta-model. The SAN meta-model we propose encompasses concepts from the OMG's Structured Metrics Meta-model (SMM) [138], the Semantic Sensor Network ontology (SSN) [134], as well as SensorML [24]. This model describes the entities in the system, data-types sent and received, as well as the events and processes happening in the system. It provides a better view and understanding of the system, instead of directly implementing it. Second, a model transformation is performed on this model. This transformation generates a set of HTTP queries, which are executed in order to configure a web-based monitoring platform. Third, using the MQTT protocol, our

monitoring platform is able to monitor the SAN, trace its execution, and perform additional analysis. By leveraging MDE for generating a monitoring platform for a CPS, this approach fosters a better reusability, understandability, and provides a safer way of structuring SANs than code-based engineering approaches.

This chapter is organized as follow: Section 4.2 first describes the SAN meta-model, Section 4.3 presents Emit, our monitoring platform, Section 4.4 presents a model transformation for adding SAN's entities into Emit and enabling their monitoring, and a use-case is presented in Section 4.5.

4.2 Sensor and Actuator Network Modeling

We propose a SAN meta-model as shown in Figure 4.2. It aims at generically representing the structural and behavioral characteristics of each SAN. From a structural perspective, our meta-model enables the specification of network entities (e.g. sensors and actuators) inside areas, and the relationships between them. From a behavioral perspective, our meta-model allows users to specify the heterogeneous data that are gathered by sensors, the communications between the network entities, and the way actuators behave after receiving data. In what follows, we detail our SAN meta-model.

4.2.1 Foundations

Related work has proposed several SAN modeling languages. We have created our SAN meta-model based on the following works: Janowicz et al. propose Sosa[92], a lightweight ontology for Sensors and Actuators. It enables the description of Sensors and Actuators, the device on which measurements are performed, as well as the results it outputs. However, since it is lightweight and generic, it is

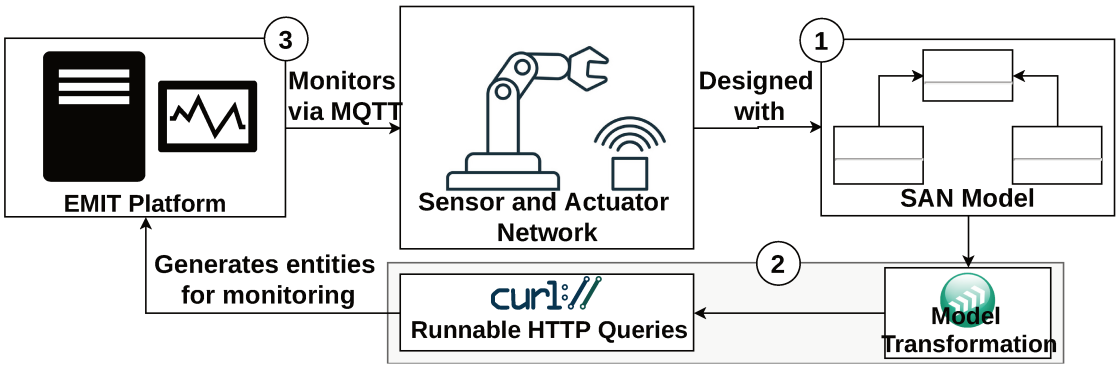


Figure 4.1: Approach for CPS monitoring in Emit

coarse-grained, offers little expressivity on the definition of the devices, and is not meant to specify processes. OpenGIS proposes SensorML [24]. It can be used for modeling many aspects of sensors networks: data-types, positions, temporal data, encoding, observable properties, processes, etc. SensorML covers most of the needs but does not enable the representation of actuators. Ben Maissa et al. design a large set of primitives to specify physical deployment of sensors/actuators, or to specify different behaviors of sensors/actuators according to their location within the network [17]. The main limitation is that the result lacks of modularity, which makes maintenance difficult. Doddapaneni et al. separately model the architecture of SAN, the low-level hardware specification of its nodes, and the physical environment of deployed nodes [56]. This kind of separation of concerns is similar to our way of modularizing design of physical and software parts when designing CPSs, but in their work they model CPSs for simulation purposes, whereas our work focuses on monitoring running systems. On the same track of enhancing design modularity, Dantas et al. design a language called LWiSSy to encourage domain experts to contribute to the design of SAN [51]. Specifically, the language is organized in three views: structural, behavioral, and optimization. Each view is to specify a distinct aspect of CPS under development by different domain experts. LWiSSy offers little expressivity on the data measured by sensors. Vidal et al. design MindCPS [194], which provides modelling primitives to explicitly model autonomic behaviors of system under development. Then, it can enable model transformations to automatically generate Java and SQL code for SAN (e.g. control loops). Compared to our SAN meta-model, MindCPS's meta-model is more complex, and is composed of built-in classes for data filtering, error handling, etc. Moreover, in terms of real-time data processing, MindCPS provides a mature solution that publishes them on the MQTT broker in order to provide real-time feedback. This feature of MindCPS is what we intend to extend for our approach in the near future. Neuhaus et al. propose an ontology for describing sensor networks [134]. Sensors can be defined with positions, measures and processes can be triggered when specified guards are validated. However, it lacks the description of *Measurands* on which measurements are performed, and a more reusable way of defining Sensors (and Actuators), by first defining their model. To model measures and persist measurements, the OMG has defined SMM [138], first presented in Chapter 2. It provides a unified way for representing the measurement information and the measures performing such measurements, without detailing the entities measured. SMM also enables the specification of relationships between measures and measurements, the definition of units of measure, dimensions, measurement scope, measurement accuracy, and so on.

We base our SAN meta-model on this related work. It is able to define sensors, actuators, as well as processes and the data types provided and used by the devices.

Furthermore the entities defined can be located in space, configured, and the expressivity provided by the meta-model enables reusability across devices, for a better maintainability. Our SAN meta-model also embeds several concepts of SMM, as relying on such standard offers a better compatibility for our approach. Our monitoring platform relies on standard web-based communication protocols, HTTP and MQTT, and also stores the data in a dedicated base for enabling on-demand data analysis.

4.2.2 SAN Meta-Model

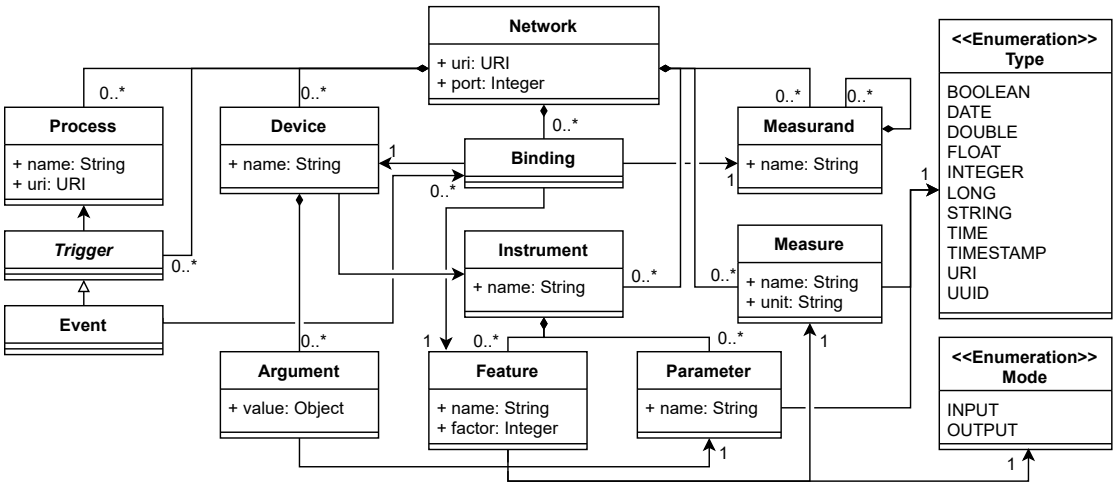


Figure 4.2: Sensor and Actuator Network Meta-model

The root of our SAN meta-model is the `Network` element. It defines the address of the server, later used by the devices of the system. `Network` enables the specification of the whole system structure through a set of `Measurand`s. Each `Measurand` can be seen as individual equipment, nested places, or zones in which instruments are installed, and performing measures on. This hierarchical structure corresponds to the characterization of physical environment used by Javier Muñoz et al. [130] .

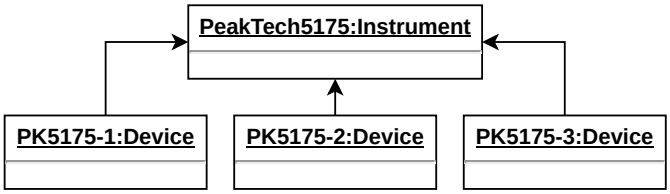
`Network` also includes a list of `Measure` elements. Each `Measure` specifies a type of measurement provided by a network `Instrument` (e.g. a sensor/actuator of the system). Both `Measure` and `Measurand` notions in our SAN meta-model are equivalent to the ones available in the *Structured Metrics Meta-model* [138]. `Measurand`s are related to `Device`s and `Feature`s using the `Binding` element. Each `Binding` links a single `Measurand` to a single `Device` and a single `Feature`, in order to define where, which entity, and how it is gathering data in the system. `Devices` define the physical sensors and actuators deployed in the system.

`Instruments` and `Devices` are closely related: the `Instrument` defines the model of the sensor (or actuator), whereas the `Device` defines its physical instance in the system. Figure 4.3a proposes an example of this concept: the instrument `PeakTech5175` is a sound-level meter. Three devices of the same sound-level meter (e.g. `PT5175_1`) are deployed. `Instruments` are defined by a `name`, a list of `Features`, and a list of `Parameters`. A `Feature` has a `Mode`. `Mode` can be either `input` or `output`, which defines the communication mode of the related `Instrument`. In fact, actuators are basic instruments that follow orders sent by a monitoring application. Thus, their related `Features` have the `Mode` set to `Input`. On the contrary, sensors send measurements to a monitoring application, hence their `Mode` is `output`. The `Parameter` category corresponds to the elements that vary among different devices of the same `Instrument`, such as identifiers of IoT devices, security tokens, etc. The `Argument` entity available in `Devices` affects a value to this parameter. All the `Devices` related to the same `Instrument` comply with its `Features`, and every `Feature` can be related to a `Measure`. A `Feature` can apply a `factor` to its related measure.

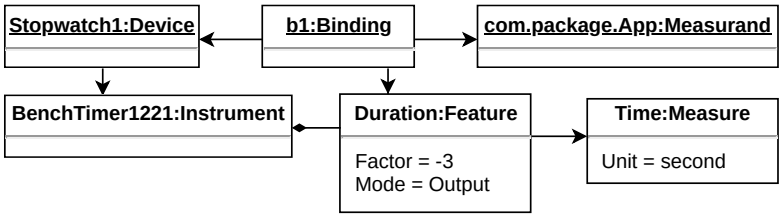
Figure 4.3b illustrates the modeling of a measurement to be performed: a remote chronometer is used to measure the execution time of a program, represented by the measurand `com.package.App`. The measuring device is named `StopWatch1` and complies with the instrument `BenchTimer1221`. This instrument has a feature `Duration`, that applies a factor of -3 to the `second` unit given by its corresponding `Time` measure, meaning that this feature records milliseconds. Finally, a binding shows that the `Duration` feature of `StopWatch1` measures the execution time of `com.package.App`.

`Processes` define computation units that can be triggered when specific events occur. The `Trigger`'s role is to specify in which situation the `Process` shall start. `Triggers` are abstract, but extended by `Events`. An `Event` is related to a `Device` and a `Feature` by the means of a `Binding`, and can launch a `Process` when the user-specified event happens. The fact that the `Trigger` class is abstract makes possible to extend it to other kinds of triggers such as background tasks or finite state machines. Using state machine using ThingML [82] is a perspective.

In conclusion, guided by a high level of abstracted design as shown in Fig 4.2, our SAN meta-model will help developers to implement their desired logic. It describes how the measurements can be started or triggered, but not how these measurements are produced, as it has no impact on how to map the SAN model to the monitoring platform. However, the reusability and adaptability of MDE would enable an extension of this existing SAN meta-model, with the descriptions of such computations.



(a) Three devices of a same instrument



(b) Feature for millisecond measurements

Figure 4.3: Excerpts of SAN models

4.3 Monitoring platform

As first presented in Section 2.2.6, MQTT is among the most popular and used communication protocol for CPS devices. Two mature monitoring platforms for MQTT devices can be considered: IBM's Watson IoT[1] and Eclipse MQTT Spy[2]. The first one is a powerful private cloud-based solution, also driven through HTTP. The second one, Eclipse MQTT Spy, offers efficient message filtering and processing service, but no HTTP endpoint for remote control and monitoring. In opposition to these monitoring platforms, we propose EMIT, a Free, Open Source, self hosted monitoring platforms for CPSs[3]. It is model driven, and provides a better modularity and adaptability than state-of-the-art platforms.

A first prototype of a monitoring application was used with power sensors measuring the energy consumption of Java programs [166]. Emit has been created using this work to automate its execution and its configuration by integrating it in our approach. Moreover, Emit can now consider all kinds of MQTT clients, with an HTTP endpoint, and analysis features through *Callbacks*. It is composed of two layers. One relies on the MQTT protocol for communicating with the devices of the CPS it monitors. The other one relies on the HTTP protocol, and is focused on configuring the entities of the CPS that Emit listens to. These web APIs has been testified by our partners from the Measure project[4] in developing advance

[1]https://www.ibm.com/cloud/watson-iot-platform
[2]https://www.eclipse.org/paho/components/mqtt-spy/
[3]https://github.com/jeromerocheteau/emit
[4]The MEASURE project. http://measure.softeam-rd.eu/related-tools

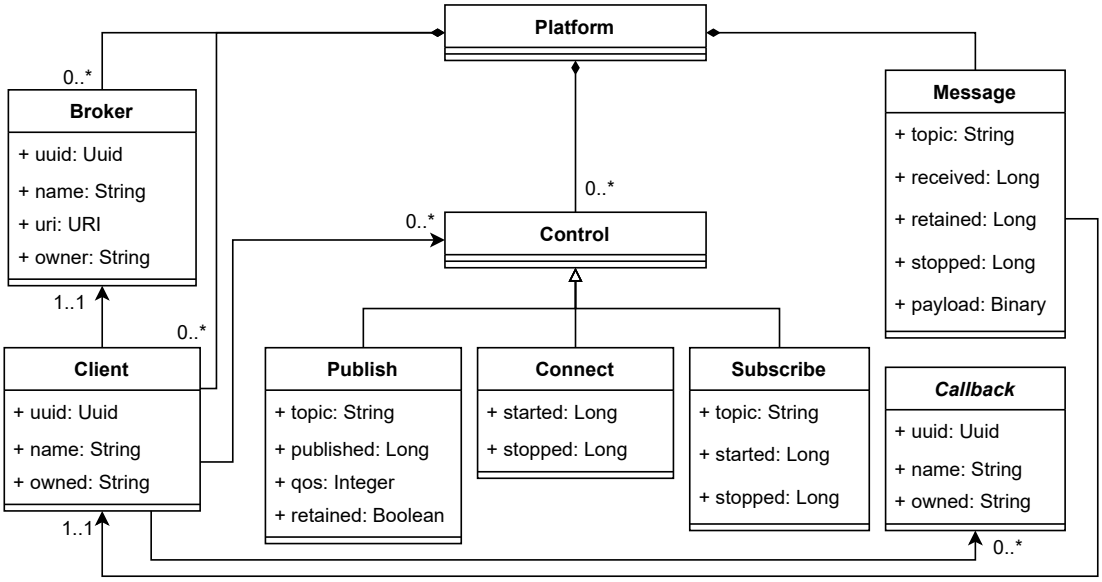


Figure 4.4: Emit Core Meta-model

web applications, such as tools for security analysis and data mining.

A bird's eye view of Emit is shown in Figure 4.4. It manages a set of `Clients` which publish/subscribe `Messages`, and a set of `Brokers` which conditionally dispatch `Messagess` to `Clients`. It enables a set of standard `Controls` over `Clients`, such as `Connect` to a broker, `Subscribe`, and `Publish` to a topic. Last, to have finer-control over clients, Emit also manages a set of `Callback`, i.e., functions to be called when clients receive messages.

4.3.1 Client Management

The first web API supported by Emit has the purpose of managing MQTT clients. This implies being able to create, update, delete and retrieve MQTT clients among the instances of `Brokers`. After providing connection settings and credentials, MQTT clients can be connected to existing `Brokers`, in order to subscribe, unsubscribe, and/or publish on specific topics. One notable feature of Emit is that it maintains an inner pool of registered clients that are trusted by Emit. Foreigner clients need to correspond to a registered client in Emit in order to be authenticated and to perform management operations via Emit.

4.3.2 Client States Control

The second web API available in EMIT enables the management of the clients states. For each client, this implies being able to: (1) Connect and disconnect the client to a MQTT broker, (2) Subscribe and unsubscribe the client to a MQTT topic, (3) Publish a message to a topic.

Those operations are performed using existing MQTT libraries (https://mosquitto.org/). All performed modifications are logged by EMIT. For instance, calling the web API that connects a MQTT client to a broker creates a new `Connect` instance in the database with the `started` property defined by the current timestamp. When it disconnects, the web API updates the instance for its `stopped` attribute with the current timestamp. This works the same way for the `Subscribe` and `Publish` entities.

4.3.3 Callback edition

Callbacks are behaviors that are triggered when a specified event occurs. In EMIT, the web API enables the creation and edition of callbacks. `Callbacks` are created by `Clients`, and trigger events when a message corresponding to this `Callbacks` is received on a topic listened by the `Client`. In EMIT, we can create four types of atomic callbacks for MQTT message processing (as shown in Figure 4.5):

- `TopicCallback` returns true if and only if the message topic matches the `pattern` attribute it contains.
- `TypeCallback` returns true if and only if the message payload can be cast to one of the `types`.
- `StorageCallback` enables the storage of a message content, into a given collection of underlying database of EMIT. Hence it means that the message persistence is programmatically defined using the callback edition services, with their attachment to a client.
- `FeatureCallback` returns true if and only if the message payload satisfies the logical condition expressed via `Type` and `Symbol`.

These atomic callbacks can be composed into more complex callbacks using `GuardCallback` for a conditional case analysis structure. Each `GuardCallback` specifies: 1) a `test` callback to check the condition; 2) a `success` callback to specify what should happen when the check passes; and 3) a `failure` callback to specify the case when the check does not pass. EMIT also enables the definition of callback when events at the user level occur.

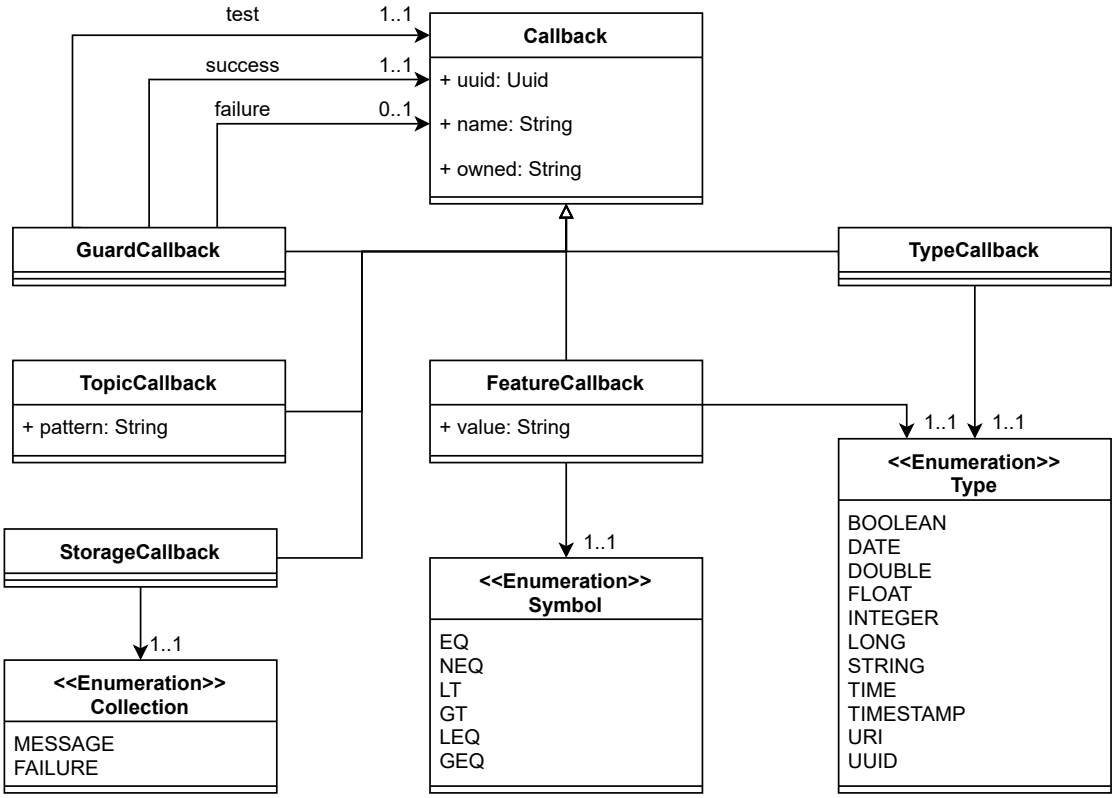


Figure 4.5: Emit Core callback meta-model

- `ConnectCallback` corresponds to the connect / disconnect state control.
- `SubscribeCallback` corresponds to the subscribe / unsubscribe state control.
- `AttachCallback` corresponds to the attach/detach state control. The duality in client state controls (e,g. connect/disconnect, subscribe/unsubscribe and attach/detach) is defined by a Boolean property labeled `enable`.
- `PublishCallback` corresponds to the publish state control, where its `message` property defines a message payload.

Finally, Emit provides a web API for MQTT messages retrieval. This returns chunks of messages for a given MQTT client, on a specified topic. The service can be customized in order to only retrieve messages in a given time-span, by giving the API temporal bounds.

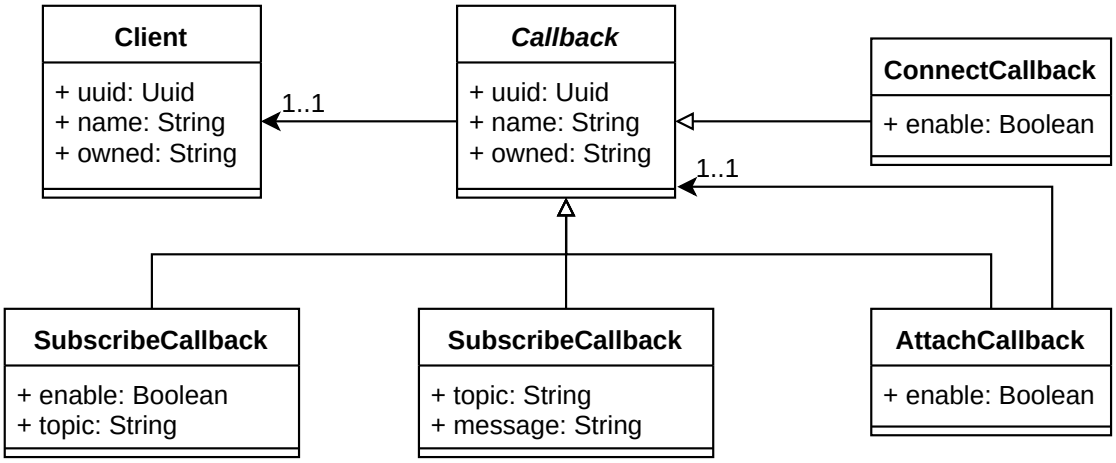


Figure 4.6: Emit Core callback meta-model

4.4 Mapping SAN models to Emit

In previous sections, we presented our SAN meta-model to provide a way to model the diversities in SAN (Section 4.2). We also presented Emit, a generic CPS monitoring platform relying on Web-APIs for configuration and management (Section 4.3). This separation of designs allows experts from both SAN modelling and SAN management to work independently, thereby increasing productivity and quality of software development. In this section, we describe a mapping between the two worlds, that allows domain experts in each world to exchange their information. Our mapping is documented in a model transformation fashion, which aims at generating entities inside Emit, hence enabling monitoring the CPS corresponding to the SAN model. In addition, since the web APIs of Emit can only be queried through HTTP requests, we briefly illustrate the corresponding HTTP queries generated out of the SAN model, that facilitate the usage of Emit.

Note that our mapping is not a **total** function, in the sense that several elements in the SAN meta-model do not have to be created in Emit to enable the monitoring of the CPS. As an example, `Measurand` elements are physically bound to the devices performing measures on, and are modelled in the SAN models. Adding them in Emit is not necessary, since we only are only interested in monitoring the `devices` performing the measurements.

Network

At the root of our mapping, we translate each `Network` element of a SAN model into a `Broker` entity for Emit. From such `Broker` element, we subsequently launch a code generation of `HTTP POST` request in order to create a virtual broker in Emit.

The idea is to enable and follow the methodology of the MQTT protocol, i.e. using a `Broker` to manage communications of clients, later generated.

Features

In the SAN model, the `Feature` elements correspond to either sensors or actuators. The `Mode` attribute defines if either the `Feature` sends data (i.e., is a sensor), or receive data (i.e., is an actuator). In both cases, Emit has to listen to the messages going from or to the `Feature`. For that purpose, SAN's `Feature` are mapped to Client entities within Emit. Later, Emit can then subscribe to topic corresponding to the mapped `Feature` in order to monitor it. This topic is generated by concatenating the `name` attributes of the network, device and binding, and using backslash as a separator. We assume that the monitored sensors and actuators are publishing and subscribing, respectively, to this topic.

Furthermore, the `Measure` to which the `Feature` is bound to indicates which datatype this `Feature` is supposed to receive (or send). A `GuardCallback` is added to this generated client to ensure the data sent conforms to the right datatype. This `GuardCallback` checks the type of the message using a `Type Callback`. Two `StorageCallback`s are then attached to this `GuardCallback`. If the type of the message conforms to the `Measure`, then the messages can be persisted in a "message" collection with the first `StorageCallback`. Otherwise, it is persisted in a "failure" collection with the second `StorageCallback`. After the creation of the client, Emit can monitor the messages sent to, or by, this client and eventually add callbacks to it according to the needs.

Events

So far, only the events attached to features with the `output` mode (i.e., events triggered when a specific value is sent by a sensor) are considered by this transformation. In fact, actuators are not necessarily controlled through MQTT, and for that reason, mapping them into EMIT is out of this work scope. This Event transformation shows that it is possible to automatically create within Emit: (1) a Callback element that corresponds to this process, and (2) a Client that subscribes to the topic on which a device broadcasts its measurement data.

The `Event` transformation has two steps. Firstly, it verifies that the `uri` of each `Process` actually refers to an existing entity within the execution environment. This can be done using software components provided as third-party libraries and which comply with the MQTT callback API provided by the Eclipse Paho library[5], used within Emit. Secondly, it transforms every `Event` bound to `output` `Feature`s into a `GuardCallback`. The test of this `GuardCallback` is defined by a

[5]Eclipse Paho. https://www.eclipse.org/paho/

`FeatureCallback`, which checks if the output of the `Feature` verifies the specified condition. The success of this `GuardCallback` is defined by a hand-made callback launching the `Process` behaviour. Such transformation provides flexibility: custom MQTT callbacks can be developed and integrated into the monitoring application later. However, it does not ensure a full control over the provided callbacks which have to be user-defined in order to launch `Process`es.

To summarize this mapping, we display in Section 4.4 a SAN model and its corresponding entities in Emit, according to this transformation.

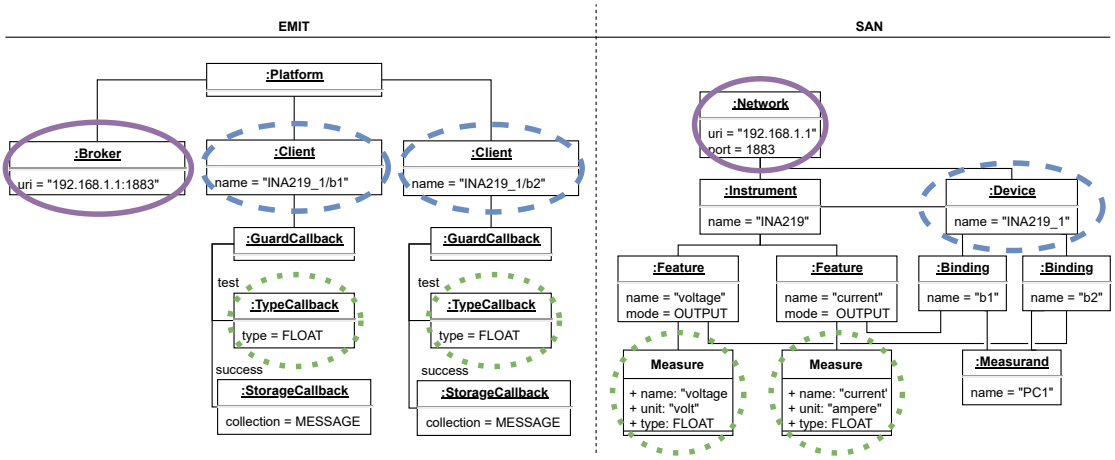


Figure 4.7: Mapping from SAN to Emit

4.4.1 Mapping from other meta-models

We previously showed how the SAN meta-model can be mapped to Emit using a model transformation. This transformation generates a set of HTTP queries that, when executed, enables the configuration of Emit. This MDE approach is easily configurable and *reusable*, as any other entry meta-model could be mapped to Emit for a remote web-based monitoring.

In the previous chapter, we mentioned existing scalability issues for model-driven approaches. When the analysis produces too many measurements, persisting the SMM-based analysis can be complicated, and the model might not fit in the memory. Emit would offer an interesting alternative to it, as instead of being modeled with SMM, the measurements could be stored in the document-based database of Emit, once sent through MQTT. This would offer a better scalability in space.

As an example, we could measure the energy consumed by the methods of a Java program, and send the measurements to Emit. Instead of tracing and modeling

```java
class C {
    void A() {
        B();
    }
    void B() {
        doSomething();
    }
}
```

(a) Java source code

```java
class C {
    void A() {
        double beginning = EnergyCheck.statCheck();
        B();
        double energy = EnergyCheck.statCheck() - beginning;
        MQTT.publish("C.A", energy, 0);
    }
    void B() {
        double beginning = EnergyCheck.statCheck();
        doSomething();
        double energy = EnergyCheck.statCheck() - beginning;
        MQTT.publish("C.B", energy, 0);
    }
}
```

(b) Java source code after instrumenting methods

Figure 4.8: Source code instrumentation of a Java program to send estimations to Emit.

the execution with SMM, the instrumentation step performed would perform measurements, and sent them to Emit through MQTT. A first model transformation would generate a set of HTTP queries from a MoDisco model, to configure Emit. Figure 4.8 shows an example of a source code instrumentation that would enable such behavior. The instrumentation first calculates the energy of each method using jRAPL, and finally sends the results on an MQTT topic corresponding to the qualified method name. The last parameter of `MQTT.publish()` corresponds to the MQTT *QoS* parameter.

4.5 Application

In this section, we apply our approach on a case study, following the three steps presented in Figure 4.1: (1) Design of the SAN model, (2) Model transformation and (3) Emit configuration by executing the generated HTTP Queries. This case study is designed for a smart cooling system of an IT infrastructure: a thermometer measures the temperature in a building, when this temperature reaches a given threshold, a cooling process starts. The infrastructure is modeled conforming to our SAN meta-model (Section 4.2) as shown in Figure 4.9[6]. We then describe how it is mapped to Emit for enabling its monitoring. The goals are: (1) To show the possibility of pipelining our technologies for developing and monitoring CPS. (2) To provide this case study as a reference that allows SAN users to define their own mappings in order to interact with Emit.

4.5.1 Modeling a case study

The IT infrastructure is composed of three locations in which measurements are performed: two floors `Floor1` and `Floor2` located inside a building `Building1`. One thermometer device (`TILM35B`) is attached to the whole building for measuring its current temperature. It refers to the `TILM35` element, which describes the measuring features it uses. We labelled it `CurrentTemp`, as it produces a temperature with the Celcius unit. A binding labelled `b3` associates together `Building1`, `CurrentTemp` and `TILM35B`. This models that the thermometer `TILM35B` is deployed in `Building1`, and is measuring its temperature (in Celcius) with the temperature sensor it embeds. Furthermore, a power meter (measuring watts) is attached to each floor for tracking their respective power consumption. They are modeled by the `PeaktechF1` and `PeaktechF2` elements, and refer to `Peacktech6226`, which describes its power measuring output device. The `b1` binding associates together `Floor1`, `PeaktechF1` and `PowerConsumption` (`b2` binds `Floor2`, `PeaktechF2` and `PowerConsumption`, respectively). Finally, an entity "`ProbingTemp`" is referring `b3`. This entity triggers a `Cooling` process as soon as the temperature measured by the `TILM35B` is too important. We modeled this SAN, as shown in the first step of Figure 4.1, the next step it to automatically generate its monitor application.

4.5.2 Mapping to Emit

In order to enable monitoring, the modeled SAN has to be added into Emit. Running the transformation (as described in Section 4.4) on the input SAN model

[6]The figure is simplified to fit the level of conciseness needed for this thesis. In particular, containment relationship of Network entities are hidden, and only entities that are demonstrated here are shown.

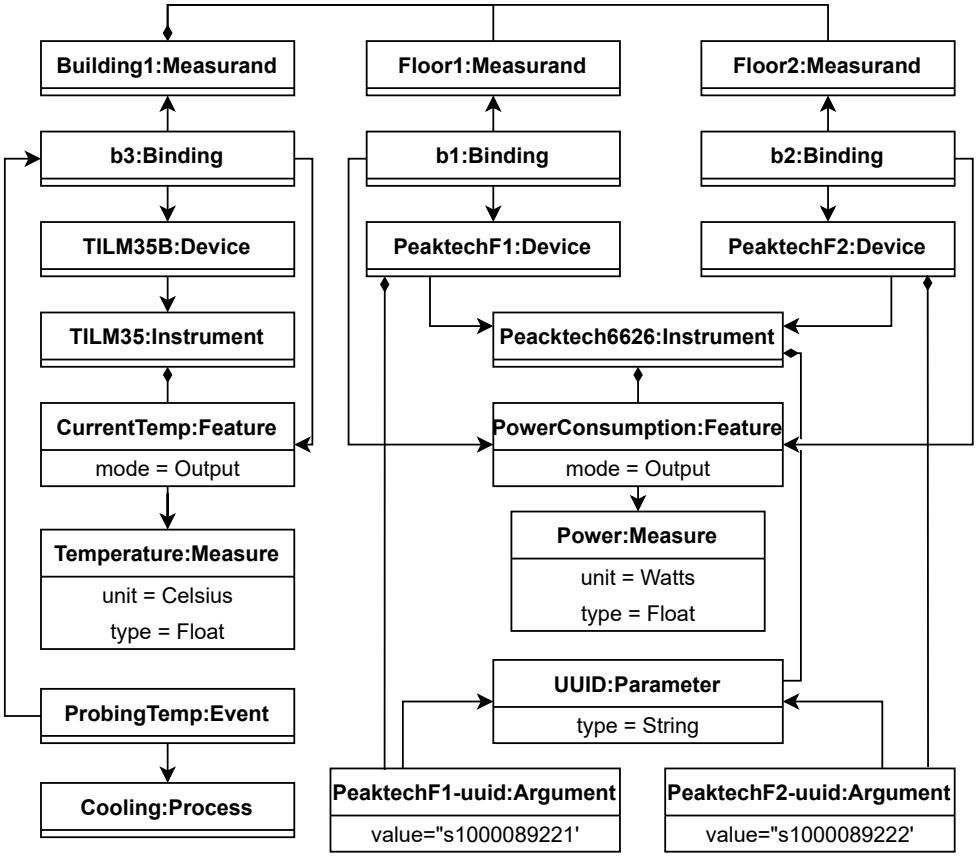


Figure 4.9: Cooling System Modeling for IT Infrastructure using SAN model

generates several entities in Emit.

Since Emit relies on the MQTT protocol (as described in Section 4.3), the first step of the transformation adds into Emit the broker used in the SAN. As shown in Figure 4.10a, a broker labeled `Infrastructure` is added to Emit, which corresponds to the root `Platform` element of the SAN model (not displayed in Figure 4.9 for readability purposes). Subsequently, it allows Emit to subscribe to different topics that this broker hosts, thereby monitoring the sensors and actuators available in the system.

The second step of the transformation maps the three devices to Emit as Clients. Figure 4.10b shows the interface of Emit that displays a client generated for the `PowerConsumption` feature of `PeaktechF1`. The user can switch between clients by using the left panel. The right panel shows different parameters of this feature that the user can edit. As we can see in the figure, `PowerConsumption` is declared as a public client, which means it can be seen by any Emit user (private client would only be seen by the user that added it). The last text field shows the

Broker in Emit, to which this client is connected. Once registered to a broker in this way, client's data is sent and monitored by Emit. This allows further analysis or computations to be performed.

The third step of the transformation maps the event in the SAN model into Emit. For instance, the `ProbingTemp` event is transformed into a set of callbacks. First, a guard is created into Emit. This guard is a callback, activated every time a message is sent by the corresponding client. The test of this guard checks if the power measured by the `PowerConsumption` sensor is above a specified threshold. A success from this guard launches the cooling process linked to the event. Figure 4.10c shows the interface of callbacks in our example and the guard generated from the SAN model.

Upon this point, the smart cooling system is added in Emit, as shown in the second step of Figure 4.1.

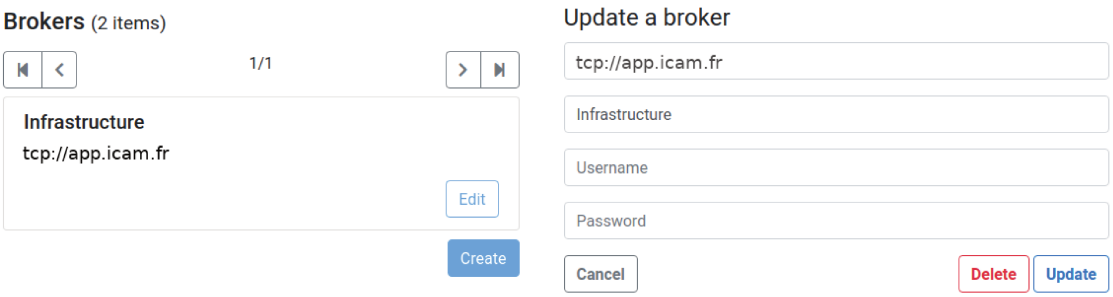
4.5.3 Monitoring with Emit

Finally, Emit can effectively be used to monitor the three clients created, as shown by the third step of Figure 4.1. This enables several functions: (1) Adding, removing, and managing more `Callbacks` for the clients. (2) Easily accessing the messages sent and received by the clients, using the RESTful API provided by Emit. (3) Providing real-time metrics with graphical display through the front-end of Emit, as shown in Figure 4.10d. Monitoring this system within Emit enables different analysis and computation on the data gathered by the sensors of the system. E.g., energy metrics and statistics could be computed using both the power-meter and timestamp values providing the energy consumption of the building during the wanted duration.

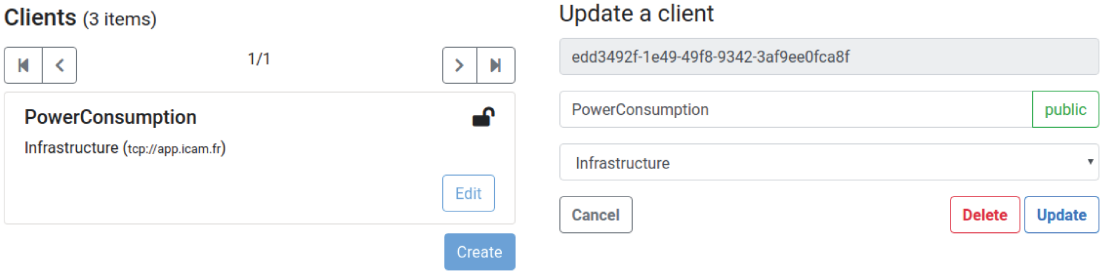
This case study shows how we generate Emit entities from our SAN model, in a real-life situation. This approach is automatized using a model transformation, ensuring a conformity between Emit's configuration, and the SAN model. Compared to standard approaches, modifying the system would not break the monitoring functions of Emit: re-running the transformation would update the entities of Emit and hence maintain the monitoring, for a better reliability and maintainability.

The transformation in this example is implemented as a Model-To-Text (M2T) transformation, and is available on Github[7]. The SAN model is defined using the Eclipse Modeling Framework [183], and the M2T implemented using Acceleo [131], which generates a set of HTTP requests in the CURL format[8].

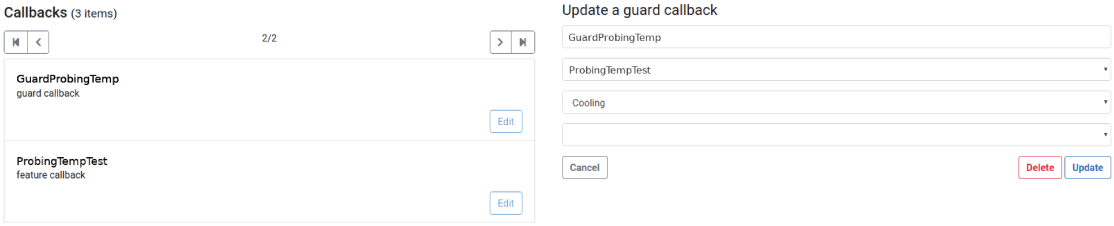
[7] https://github.com/veriatl/emit-san-metamodel
[8] https://curl.haxx.se/



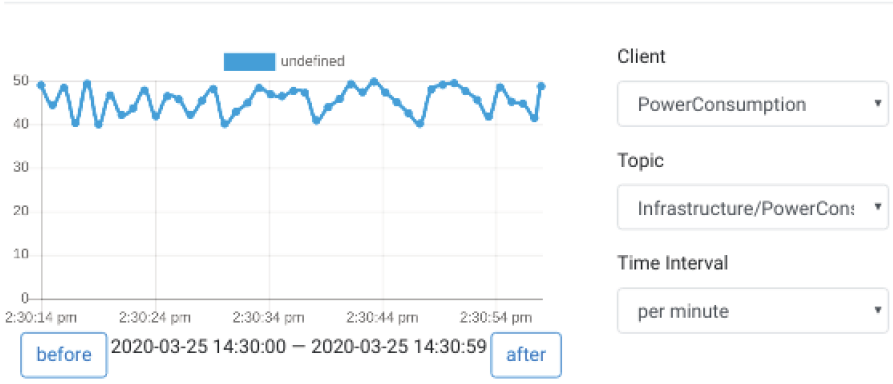
(a) Broker view



(b) Client view



(c) Callback view



(d) EMIT monitoring view

Figure 4.10: Screen captures of Emit running

4.6 Conclusion

In this chapter, we present a model-driven approach to monitor and trace a specific type of CPS: Sensor and Actuator Networks. We introduce a meta-model of SAN, enabling the co-design of the structural and behavioral parts of such network. We also design and implement an open-source Web-API, namely Emit, for monitoring sensors and actuators networks, and providing an endpoint to the information infrastructure for running analysis and computations on the data gathered from the network. The generic meta-model of Emit as well as the simple HTTP-based configuration endpoint makes it easy to use for any kind of remote dynamic analysis, even software. We orchestrate our SAN meta-model and Emit via model transformation to allow them work hand in hand and apply our whole approach on a smart cooling system case study.

The result shows that by leveraging MDE for generating an MQTT-based SAN monitoring application, we can foster reusability of CPS design. Emit can be used to monitor the CPS and persist its execution traces in its database for further analysis. The abstract view of the system eases its understandability, providing a safe way of structuring the SAN network. Furthermore, the standard format provided by EMF enables reusability and maintainability, for multiple purposes such as refactoring, refining, code generation. Finally the mapping used for transforming our SAN model to the Emit model can be easily customized, in order to adapt to the client's needs.

Chapter 5

Trace-based energy estimation

5.1 Introduction

As explained in Section 3.3 energy consumption has become an important concern in the domain of software engineering during the past decade [144]. 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 [39, 100, 135, 150, 158]. In particular, many tools have been developed for measuring and estimating the energy consumption at the software and middleware levels [25, 43, 116, 136, 163, 180]. Existing approaches from literature address the energy consumption of general-purpose programming languages instructions, e.g. based on LLVM IR [77], Android Bytecode [80], or System call traces [2].

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 system, since a significant part (usually most) of the consumption goes into software-driven physical devices [160, 172]. 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 [144, 152].

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 [3]. 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 [15, 44, 60, 121, 185].

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 chapter is organized as follows. Section 5.2 proposes a running case to exemplify the approach. Section 5.3 outlines the EEL abstract and concrete syntax, and its semantics. Section 5.4 experiments on the approach. Section 5.5 concludes this chapter.

5.2 Running Example

We illustrate the chapter with a running example where a developer wants to build a small CPS on top of Arduino[1].

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

[1] https://www.arduino.cc/

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 here, 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 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 [26], AToM3 [53], Ptolemy [35], or ModHel'X [81], and generate the

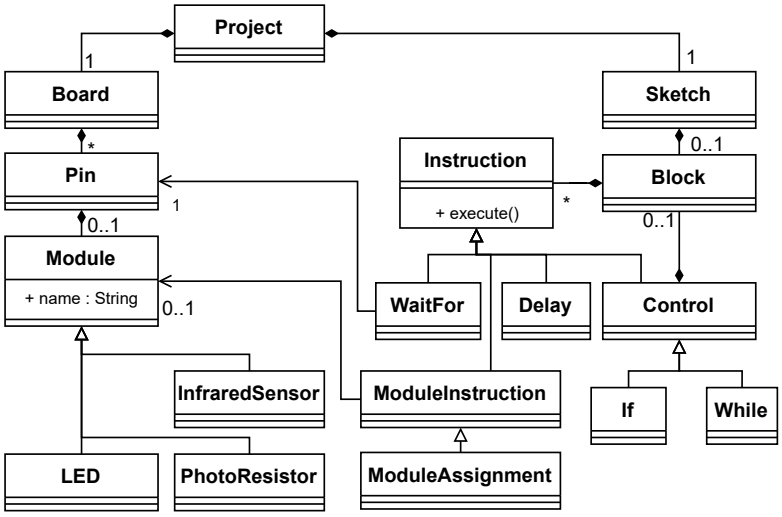


Figure 5.1: Excerpt of the ArduinoML meta-model

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.

Due to the popularity of Arduino, several model-driven approaches have been proposed to simulate and generate code for Arduino systems [26, 72]. In this work we rely on the xDSL *ArduinoML* integrated to the GEMOC Studio, based on standalone development environment Arduino Designer[2]. *ArduinoML* is an xDSL for representing structural and behavioral aspects of Arduino systems. Figure 5.1 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 5.2 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.

5.3 Energy-Estimation Modeling

5.3.1 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

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

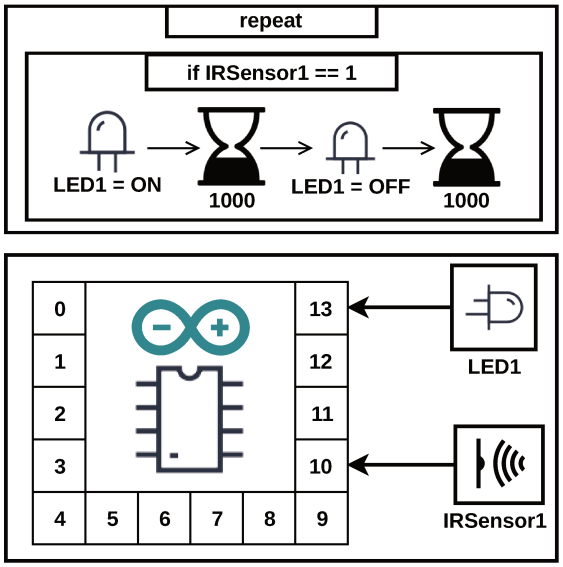


Figure 5.2: ArduinoML model

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.

The EEM in Listing 5.1 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

```
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 5.1: Excerpt of an EEM for platform made of an Arduino UnoR3, a LED actuator, an infrared sensor

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 5.1). 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 `Module Assignment` 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.

5.3.2 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 [64], is defined in Listing 5.2, and an excerpt of the abstract syntax is available in Figure 5.3. We discuss later in Section 5.3.5 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.

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

```
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 5.2: Excerpt of the EEL concrete syntax

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

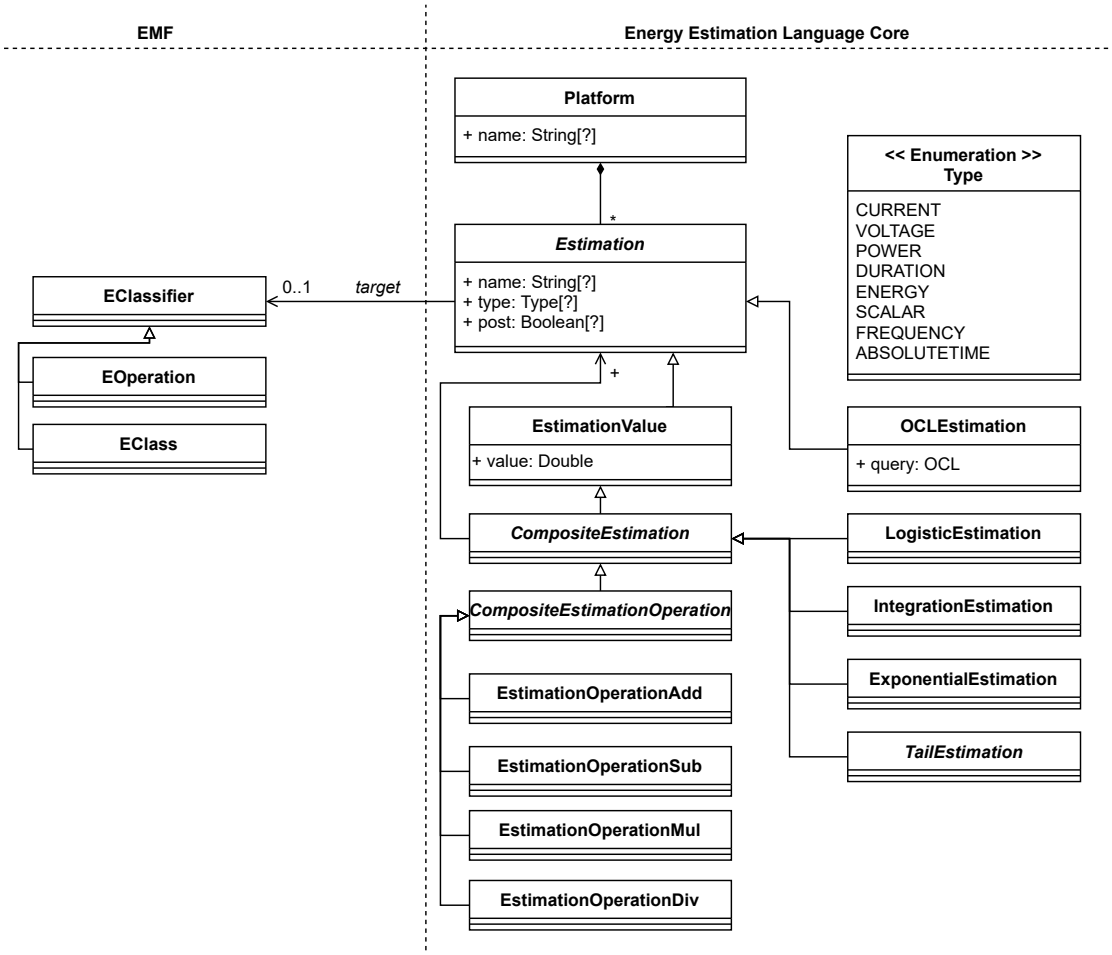


Figure 5.3: EEL abstract syntax

different flavors of OCL. For instance uncertainty-aware OCL [122] 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* metaclass. 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 5.2). 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 5.4.1).

Finally, a special estimation *absoluteTime* is used to refer to the clock value,

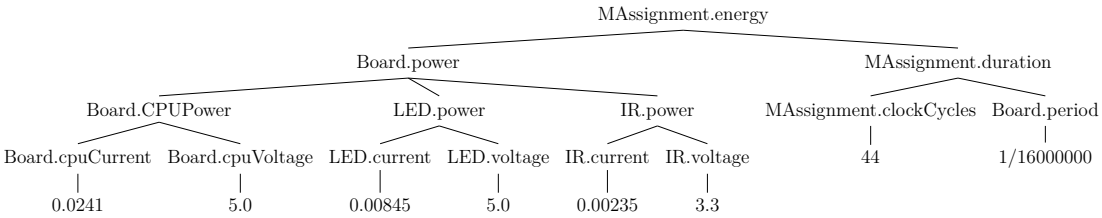


Figure 5.4: Evaluation tree for the `ModuleAssignment.execute()` operation defined in Listing 5.1

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 5.3.5.

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 5.4 shows the estimation for a `ModuleAssignment.execute()` operation, in a tree shape, based on Listing 5.1. 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.

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 5.1, 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 5.3 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.

5.3.4 The Energy-Estimation Modeling Process

EEL is meant to be used in the process illustrated by example in Figure 5.5. 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 They can be derived e.g., by executing the simulator on a set of

benchmarks (4), by different trace synthesis methods such as the ones presented in section 3.2 or chapter 4, 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. 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 5.2, 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 5.1 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 5.6. Instead of actively checking this sensor, the Arduino is put to sleep until the sensor detects a change.

```
1  Platform "DifferentLEDs" {
2      // LED L-53MBDL
3      LED.voltage = 5
4      LED.53MBDL_current = 0.00845
5      LED.53MBDL_power = LED.voltage * LED.53MBDL_current
6
7      // LED L-7113ID
8      LED.7113ID_current = 0.00765
9      LED.7113ID_power = LED.voltage * LED.7113ID_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 5.3: Excerpt of an EEM for a platform with different types of LEDs

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.

5.3.5 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 5.5 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),

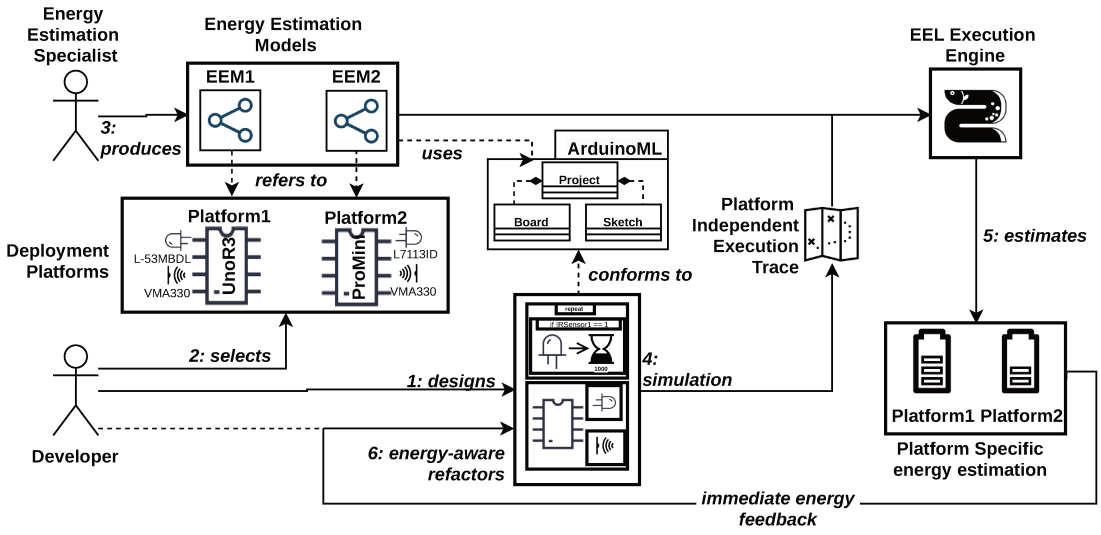


Figure 5.5: Process for the estimation of Arduino energy consumption at design time

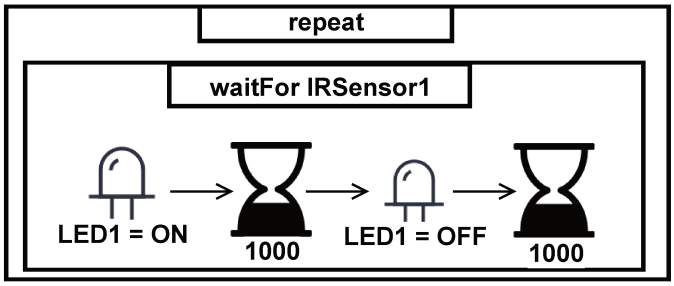


Figure 5.6: Updated Behavioral part of the ArduinoML Model

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 [94].

Non-determinism. An EEL model estimates the consumption of each given execution trace of the system. If the system 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, 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 [122] 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.

5.3.6 Implementation Details

We implemented the editor and evaluation component of EEL as an extension to GEMOC Studio. GEMOC Studio is a language and modeling workbench for model design and execution [16, 27, 29]. 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 [26]. This engine is dedicated to operational semantics directly written with Java, Xtend [59], or Kermeta [94]. 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:

- `initialize`: is called before the execution, and performs the loading of the model to be executed

- `beforeStep`: is called before executing operations annotated with `@Step` in the operational semantics.
- `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.

5.4 Evaluation

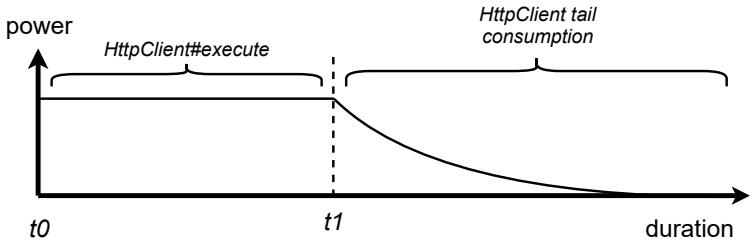
In this section we evaluate our approach against the research questions that motivated this work. The first part of this evaluation shows how EEL can model existing domain-specific energy estimation approaches from literature, answering **RQ1**. The second part presents a workflow where we use EEL to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

5.4.1 Expressiveness

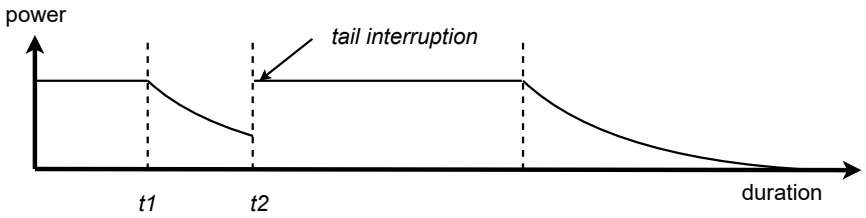
In this section we evaluate the expressiveness of EEL, i.e. its ability to model energy-estimation approaches existing in the literature. Since energy-estimation profiles for xDSLs are not currently available or not using Model-Driven Engineering [164], we select from literature well-known *per-instruction 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 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 [80]. In previous work, they show how to calculate per-line energy consumption [113], 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 5.7a 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



(a) Tail energy consumption



(b) Tail interrupted

Figure 5.7: Tail consumption behaviors

eventual interruptions of the tail consumption. The Expressiveness of EEL allows us to model such event. For instance, Figure 5.8 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 5.7a 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 5.7b, to the absoluteTime at which it started last, *t2* in Figure 5.7b. 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

```
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
```

Figure 5.8: EEM for Android HttpClient

```
1 call#execute.energy = 1.25265
2 op#execute.energy = 1.95631
3 memload#execute.energy = 0.95351
4 ...
```

Figure 5.9: EEM for LLVM IR

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 [32], for instance).

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 [77]. For such low-level consumption models, EEL can simply list the energy consumptions of each LLVM IR instructions as shown in Figure 5.9. 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 [2]. 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 Figure 5.10. 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.

```
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
```

Figure 5.10: EEL Power estimation model example for system calls

5.4.2 Estimation Accuracy

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 5.2 and 5.6.

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 5.2 or Figure 5.12) 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*.

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. Figure 5.11 visualizes some of those curves. 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, Figure 5.11a shows an average power of 122.5 mW for the Arduino UnoR3 board in running state, thus we model this power consumption with EEL, and attach it to the `Board` meta-class of ArduinoML.

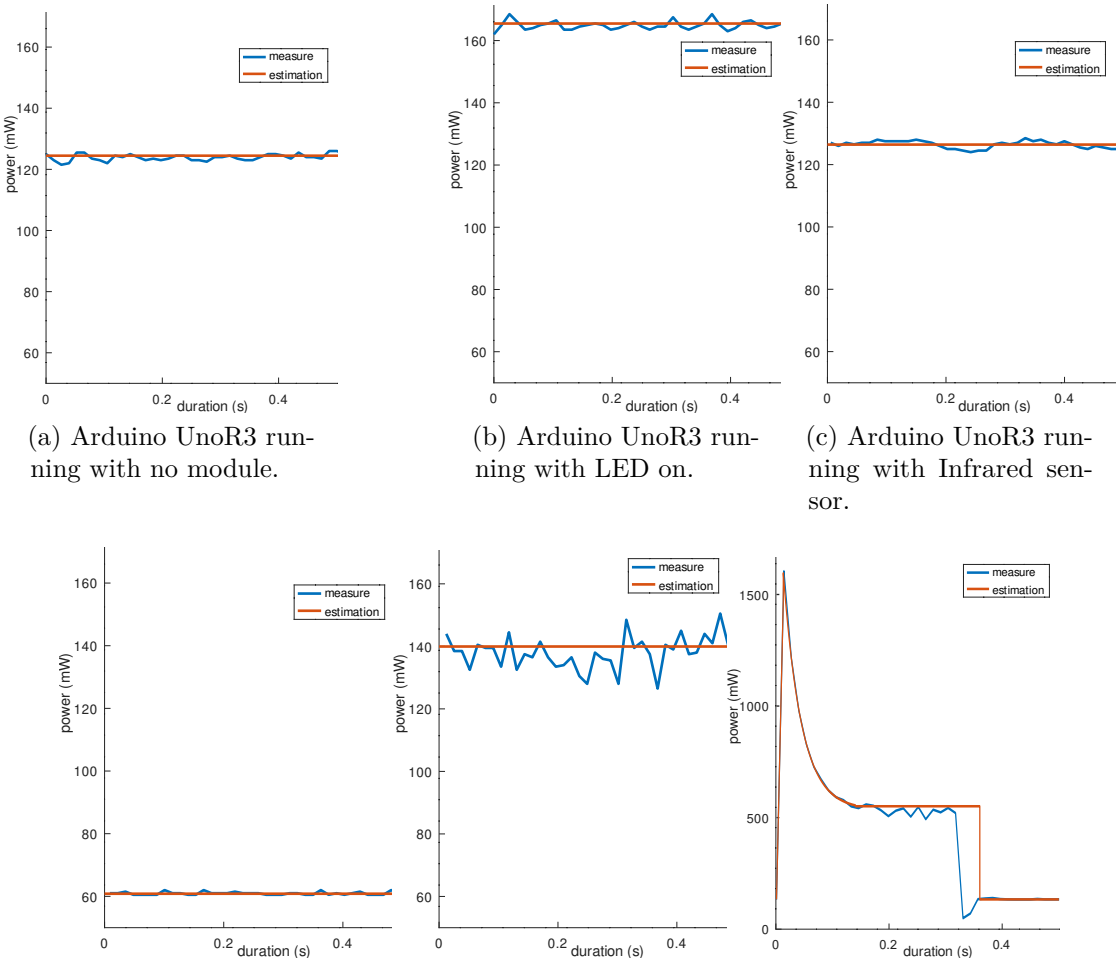
To measure the consumption of a Module, we need to compare the consumption of the platform with and without this Module activity. For instance, to estimate the consumption of the LED, we subtract the consumption of Figure 5.11b from Figure 5.11a.

Some meta-classes require more reasoning in order to be properly estimated and modeled with EEL, e.g. the photoresistor in Figure 5.11e and the servo motor in Figure 5.11f. In fact, the power consumption of the photoresistor is not constant, but depends on the intensity of the light it measures. 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. 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 5.1. In each of the plots from Figure 5.11 we display, in addition to the energy consumption measured on the platform, the power estimation curve modeled with EEL (in red).

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 have very similar shape than Figure 5.11, 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.



(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 5.11: ArduinoML benchmarks used for building Arduino UnoR3 EEM

ArduinoML Model Estimation

We use the EEM model just defined to estimate the energy consumption of the two sample ArduinoML models (Figure 5.2 and Figure 5.6, respectively labeled *IfTempo* and *WaitForTempo*) and of a third ArduinoML model (Figure 5.12), 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[3]. 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.

Table 5.1 shows the results of the estimations. The energy estimation for the UnoR3 platform is decomposed into the first four rows. The first row corresponds

[3]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

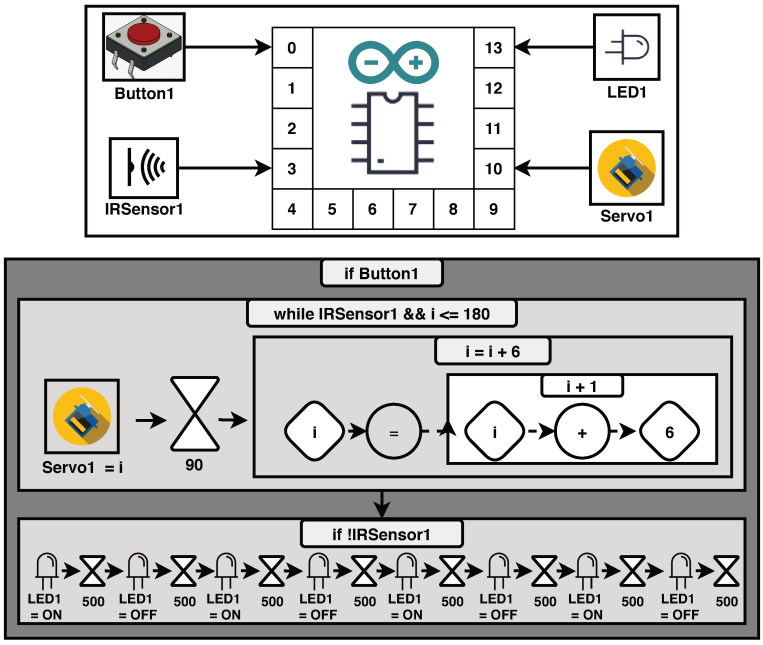


Figure 5.12: ArduinoML model of an automatic door

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.

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 [131] 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 ArduinoIDE[4].

[4] https://arduino.cc/en/main/software

	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%

Table 5.1: Comparison of the measures and estimations for the Arduino models

The Measurement column of Table 5.1 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. A power consumption plot of the system, and the estimation is available in Figure 5.13.

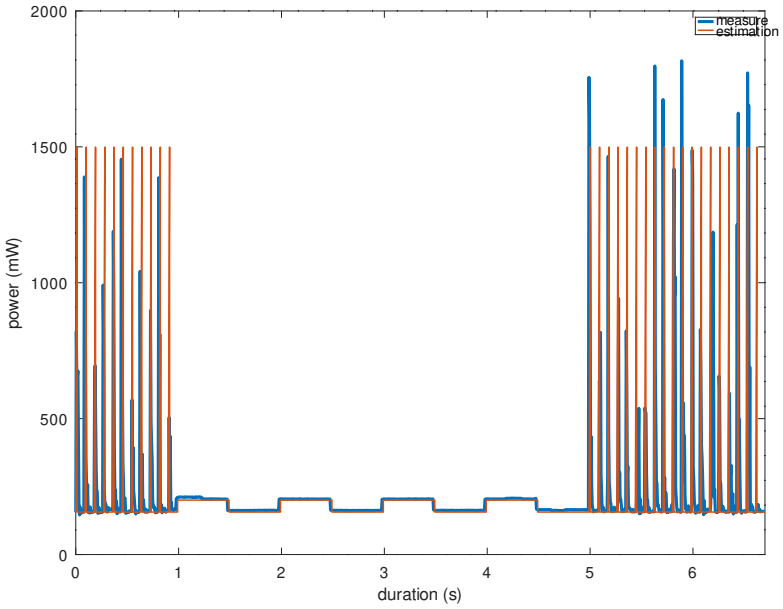


Figure 5.13: Power consumption plot of the Arduino system, and its estimation

Discussion

As shown in Table 5.1, 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 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 V port, which is wired to a second Arduino that outputs a regulated 5 V. This second Arduino measures the c urrent 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 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.

5.4.3 EEL and Emit

To conclude this evaluation we propose to use Emit to perform energy estimation with EEL. As described in Chapter 4, Emit can be used to monitor networks of sensors and actuators through the MQTT communication protocol. It persists communications in a document-based database, that can be queried and used as an execution trace of the CPS.

We reuse the CPS defined in Section 5.4, featuring a servo motor, a LED, a button and a button. In order to be externally monitored, we consider that the Arduino board embeds a ESP8266 chip that communicates, through MQTT, of the state of the modules in the CPS. This enables monitoring through Emit. Each module communicates to its specific topic, simply labelled by its name in Figure 5.12. According to the Emit's meta-model defined in Figure 4.4, we define an EEL model to estimate the energy consumption of the monitored system, when deployed on the devices specified in Section 5.4.2. This EEL model attaches energy estimation formulas to the concepts available in Emit.

For this EEL model, the language it is based on changes (i.e., Emit instead of ArduinoML), but the platform is the same one. We detail the EEL model. Emit offers little expressivity besides enabling the modeling of MQTT concepts. In the EEL model, we attach the voltages of the modules to the root `Platform` element. All execution traces are available in the `Message` entities. Messages have a MQTT topic, a corresponding client, a reception timestamp and a payload. We consider for this use-case that each module has its own MQTT topic for communicating. As an example, the "Servo1" servo motor in Figure 5.12 acts as an actuator where the rotation instructions are sent on the "Servo1" MQTT topic, by the Arduino board. Same for the LEDs, where a boolean is sent to their corresponding topics when they are lit on and off.

We detail the EEL model available in Figure 5.14. The voltage and current consumptions of the main modules used in the platform are attached to Emit's `Platform` element. The board and infrared sensors power consumption are constant.

We query the number of infrared sensors deployed by querying the number of MQTT topics monitoring infrared sensors in the model.

For this use-case, we consider that the execution trace is modeled as a set of MQTT messages, ordered by reception timestamp. Using EEL, we estimate the energy consumption of the system between each message reception. The duration between each message is computed at line 21 in Figure 5.14. This duration is then multiplied by the system's power to compute the energy consumed since the previous message. This energy consumption depends on the amount of LEDs turned on. Each time a message is received by Emit on a LED's topic, the amount of LEDs turned on is updated, as shown line 25. Finally, when the Arduino performs a servo motor's rotation, it sends the value of the rotation on the corresponding servo motor's topic. This message is thus used in line 19 to compute the energy spent by the motor for rotating.

We consider the Emit model available in Figure 5.15 as an execution trace for performing this estimation. This model covers a subset of the execution scenario presented in Figure 5.13. First, the button is pressed. This corresponds to the `Message` labeled "1" in Figure 5.15. Then the arduino controls the servo motor to perform three steps of the rotation, visible in the three messages related to the "Servo1" client. Then, the Infrared sensor detects an obstacle in `Message` 5. The rotation is put on pause, and finally the LED blinks during one second. Applying the EEL model to this Emit model would produce the energy estimations detailed in Table 5.2, for each message in the trace, for a total energy consumption of $0,19919924$ Joules.

Message	**Energy (J)**	**Description**
1	0	the time elapsed since the beginning of the execution is 0.
2	0.00805608	energy spent by the servo motor's rotation.
3	0.01959903	energy spent by the servo motor's rotation, plus the energy consumed by the system during 90 ms.
4	0.01959903	same as previous estimation.
5	0.0025651	energy spent by the system during 20ms.
6	0	the LED is turned on, but no time elapsed since the last message.
7	0.0852525	energy spent by the system, with a LED on, during 500ms.
8	0.0641275	energy spent by the system with the LED off, during 500ms.

Table 5.2: Energy estimation when applying EEL to fig. 5.15

```
1  Platform "Arduino_EMIT" {
2      Platform.voltage = 5.0
3      Platform.current = 0.0241
4
5      Platform.NumberOfIRSensor = Client.allInstances() -> filter(c |
           c.name.contains("IRSensor")) -> size()
6      Platform.IRSensorVoltage = 3.3
7      Platform.IRSensorCurrent = 0.00235
8      Platform.IRSensorPower = Platform.IRSensorVoltage *
           Platform.IRSensorCurrent * Platform.NumberOfIRSensor
9
10     Platform.LEDCurrent = 0.00845
11     Platform.LEDVoltage = 5.0
12     Platform.LEDPower = 0.00845 * 5.0
13
14     Platform.idlePower = Platform.voltage * Platform.current +
           Platform.IRSensorPower
15
16     Message.isLED = self.topic.contains('LED'))
17
18     Message.isServoMotor = self.topic.contains('ServoMotor'))
19     Message.servoEnergyConsumption = if (Message.isServoMotor) int(0, 0.002 *
           Message.payload, 22.378*5*x) else 0
20
21     Message#onReceipt.timeElapsedSinceLastMessage = self.received -
           Message#onReceipt.lastMessage
22
23     Message#onReceipt.energySpentSinceLastMessage =
           Message#onReceipt.timeElapsedSinceLastMessage * (Platform.LEDPower *
           Message#onReceipt.numberOfLedsON + Platform.idlePower) +
           Message.servoEnergyConsumption
24
25     Message#onReceipt.numberOfLedsON = if (Message.isLED) if (Message.payload
           = 1) Message#onReceipt.numberOfLedsON + 1 else if (Message.payload =
           0) Message#onReceipt.numberOfLedsON - 1 endif else
           Message#onReceipt.numberOfLedsON endif
26
27     Message#onReceipt.lastMessage = self.received
28 }
```

Figure 5.14: EEL model for estimating Emit traces

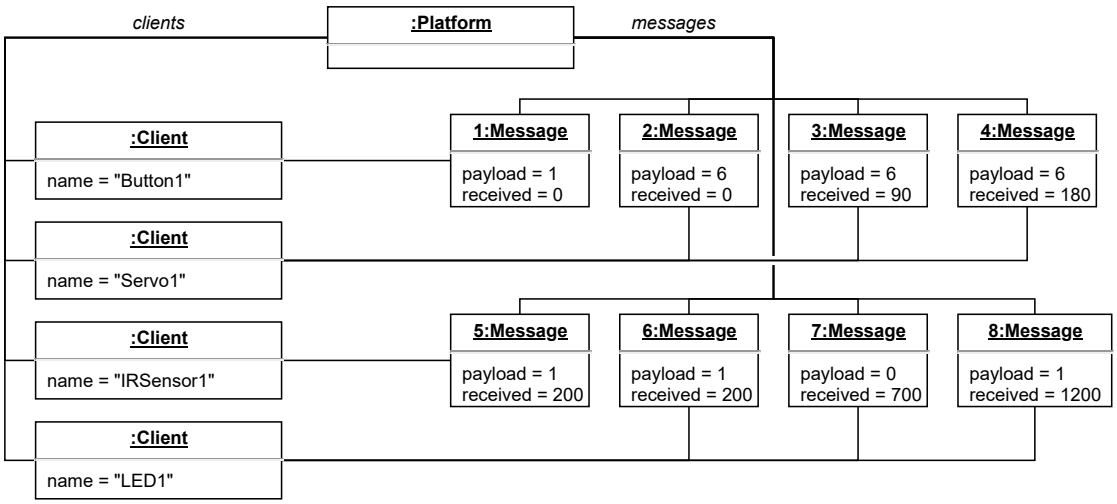


Figure 5.15: EMIT model used as an execution trace to perform an energy estimation with EEL.

5.5 Conclusion

In this chapter we presented an approach that relies on execution traces 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 chapter we focused on the syntax, semantics, and integration of the language with the modeling workbench, several improvement points can be raised. First, being able to automatically produce EEL models from sets of measure would greatly facilitate the energy specialist's role. Second, being able to re-use EEL models more easily could be a useful improvement. 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`". Third, 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. Finally, EEL could be integrated in other environments than GEMOC Studio. In fact, environments used by CPS engineers such as Simulink or Capella could benefit from EEL.

Chapter 6

Conclusion and Perspectives

6.1 Synthesis

Dynamic analysis is an important technique for asserting of the validity of software and systems. It is performed during the execution, and is able to detect issues that usual static analysis techniques would not see. In the domain of Model-Driven Engineering, software and systems are designed and generated using models. Models often represent the static aspects of the application designed, but suffer from a lack of dynamic analysis possibilities. This is especially problematic in a context of energy-aware engineering, where dynamic analysis is important to estimate and optimize the platform's energy consumption.

In this thesis, we tackle this lack of dynamic analysis options in the domain of model-driven engineering, and apply it to energy-aware engineering. We first present an approach for injecting execution traces inside models. Source code is instrumented, hence enabling the tracking of the execution. The generated traces are then injected into the initial source code model. We use this model for multiple purposes exploiting the reusability of MDE.

We first enhance the model execution by adding energy-measuring probes inside the program. Using these additional dynamic information we can model the energy consumption of the Java program, and use it as a starting point for performing energy-aware refactoring. Furthermore we rely on the execution traces injected inside the model for performing regression test selection. Regression test selection reduces the duration of the testing phase during the engineering of a software application.

If this lack of dynamic aspect is an impediment to software engineering, it is also problematic for cyber-physical systems (CPS). As an answer to that, we also propose an automated model-based approach for generating monitoring platforms of CPS. A CPS is designed using a dedicated meta-model. A model transformation takes models of a CPS as an input, and generates a monitoring configuration. When executed, this configuration defines the entities to be monitored in a web-based monitoring platform, Emit. The monitoring platform can then be used to monitor and perform dynamic analysis on the running CPS.

If these approaches can be used to trace the execution of software and systems, an additional contribution is necessary for enabling generic energy estimation of executable languages. To this extent, we propose a domain specific language (EEL) for performing energy estimation on executable models. EEL attaches energy-related concepts and energy estimation formulas to the elements of executable languages. Energy estimations can be calculated using execution traces of languages decorated with EEL. We propose several use-cases in which we estimate the energy consumption of Arduino models, as well as CPSs monitored with Emit.

To summarize this thesis, model driven dynamic analysis can be complicated, especially when models only consider static aspects of the system they design. Our

answer to this problem is to add dynamic aspects into existing static models. We propose an approach that gathers execution traces by running an application, either through simulation or code generation, and inject the traces back into the model it has been designed with. Models of dynamic aspects are an effective approach for estimating energy consumption, which is a crucial information for performing design choices aimed towards energy efficiency. Furthermore, models are effective tools for refactoring, modernizing and optimizing software and systems. Adding energy-related aspects to them can hence be an essential criterion in performing energy-aware model transformations.

6.2 Limits

Several limits were encountered during the research presented in this thesis, which should be considered in future works. We describe them in what follows.

6.2.1 Scalability

A first limit encountered early in this thesis (i.e., Section 3.2) is the lack of scalability of models, both in time and space. In fact, standard models persistence layers (e.g., XMI for EMF) are limited by the size of the system's RAM. Thus, very large models (with several millions of elements) might not fit in the available memory. This issue has been encountered several times in this thesis, either by using MoDisco to reverse engineer large Java projects (e.g., Hadoop), or when tracing the execution of complex projects (e.g., Soot). Using a more scalable persistence layer, such as NeoEMF or CDO, improves this scalability in space, however it hampers the scalability in time scalability, mandatory to perform dynamic analysis in reasonable time. If sending data measured to Emit through MQTT at runtime can help measurements dynamically gathered to scale, the static model can still remain problematic for very large source codes (hundred thousand classes).

6.2.2 Incrementality

An other limit encountered in Chapter 3 concerns the incrementality when creating models. Building a model for dynamic analysis is a costly operation: building a static model takes time, and so does running the program and inject execution traces in it. When the program is modified, our current approach has to rebuild the full model from scratch. An incremental approach would only modify the parts of the model that are impacted by the changes in the source code, instead of the entire model. Doing this should greatly improve the time performances of our

model-driven approach. Thus, it could balance the loss of speed due to the usage of a more scalable persistence layer.

6.2.3 Accuracy

This limit concerns the energy measurements, and has been problematic in Section 3.3. Accurately measuring the energy consumption of a software is a complicated task: state-of-the-art energy measurement tools, such as jRAPL, rely on hardware counters, making accurate measurement for software application hard to provide. In opposition, power estimation and measurement tools, such as PowerAPI, estimate the power consumption of processes, defined by their Process ID (PID), at a specified sampling frequency. The estimations provided by such framework are accurate for processes running for longer durations (more than a second), however estimating short running processes (few microseconds) is complicated, as the duration of the measured process might fit between two power measurements. This also results in a lack of accuracy.

6.2.4 Expressiveness

Emit enables the monitoring of cyber-physical systems through the MQTT communication protocol. Emit provides dynamic analysis features, that can be defined in its web interface, using *Callbacks*. Callbacks are functions, attached to specific topics, that are triggered when a message on this topic is received, and validates the guard of the callback. The current implementation of Emit provides little expressiveness in the manners of defining callbacks. Only simple callbacks can be defined, mostly relying on guards comparing the content of the messages with an expected value. Furthermore, the callback functions are also limited: storage in the database or message expedition. This limits the possibilities of dynamic analysis with Emit.

Furthermore, in Chapter 5 we introduced Eel, a DSL for specifying energy estimation, meant to be attached to executable languages. If Eel performs well for defining energy estimation, its expressiveness could be improved, especially for pragmatic purpose. As an example, being able to reuse already defined estimations for different platforms/meta-models to reduce copy-and-paste.

6.3 Perspectives

We mention along this thesis possible evolutions and perspectives for our work, as answers to the limits previously mentioned. In what follows we sum up these perspectives.

6.3.1 Evolution of MoDisco

This first perspective concerns MoDisco. This framework has been developed in 2011, and is no longer maintained in the NaoMod team. Extending MoDisco to enable dynamic analysis would be an interesting perspective. Our work that relied on MoDisco was focused on (1) tracing down the execution of a Java program and (2) performing energy measurements. However, proposing a high-level interface for MoDisco enabling the definition of any type of dynamic analysis for Java program would be extremely useful. A domain specific language, or a dedicated Java library, could be used to specify the dynamic analysis to be performed, and the metrics would be modeled using the Structured Metrics Meta-model, thus fostering reusability. An approach similar to Section 3.2 could be used.

Furthermore, another important evolution should be to consider scalable persistence layers. The current implementation targets the Eclipse Modeling Framework standard: XMI. MoDisco is able to reverse engineer massive projects such as Apache Hadoop, however the models produced cannot be loaded and used on standard computers due to a lack of resources. Converting XMI models to scalable persistence layers is complicated: the classes defined by the meta-model have to be generated according to the persistence layer, thus a useful addition to MoDisco would be to propose the usage of other persistence layers than XMI.

6.3.2 Hybrid model-driven RTS

This second perspective is a consequence of the previous one. Building the impact analysis model, as depicted in Section 3.4, is a long operation, depending on the complexity of the analyzed program. Due to the scalability issues described above, modeling the behavior of complex applications might not be possible with our fine granularity level (e.g., statement-level granularity). In such scenario, being able to choose the granularity at which the RTS is performed would be a relevant addition. A coarser-grained impact analysis would certainly select more test cases to be executed, but would also be faster to perform, and the impact analysis model would contain less elements. Furthermore, recent work showed that the benefits of a faster impact analysis phase can even outclass the benefits of a smaller test selection. This difference of granularity may be studied in the case of a model-driven approach for RTS.

6.3.3 Automated EEL model definition

EEL enables the definition of per-instruction energy estimations. Defining EEL models is a tedious and error-prone task: it requires specific tooling for energy measurement, as well as knowledge about the energetic properties of software and

systems, and eventually dynamic analysis skills. Automating the definition of EEL models would greatly improve the quality of life of developers: relying on a specialist of energy estimation would not be necessary anymore. However, this is complicated as EEL is meant to decorate any executable language.

An automated approach could consist in running specific instructions of a language several times, while putting parts of the platform under variable stress. Results produced could be used to infer the energetic properties of the language, when executed on the analyzed platform.

An other approach could be to rely on AI. Several EEL models, and their corresponding platform properties, could be used as learning sets for a neural network (or even a linear regression model). Once trained, providing the platform properties, the AI could propose a corresponding EEL model. This is feasible for one executable language. However it would require a lot of EEL models, which are complicated to define in the first place.

6.3.4 GPL energy estimation with EEL

EEL enables the decoration of DSL instructions with energy-related concepts. However, this approach could also be applied to General Purpose Languages (GPLs), such as Java or C++. These languages are not directly executed, but compiled instead to a lower level representation (e.g., Java byte-code), which is later interpreted by an engine. Existing work by Bugra M. Yildiz et al. proposed a meta-model for the Java byte-code [204]. Such meta-model could be decorated with EEL, and Java programs compiled to byte-code could be estimated dynamically, using this EEL model. Providing accurate energy estimation of Java applications, based on fine-grained instruction-based estimations, is complicated. Furthermore, performing this using EEL should be thoroughly evaluated, but would nevertheless provide interesting results.

6.3.5 Energy-aware source code refactoring

This perspective leverage model-driven engineering for energy efficiency. MoDisco is a popular tool for modernization and refactoring of Java programs. Refining MoDisco models and generating the code from the refined model is an effective approach for program transformation. Many energy consumption anti-patterns have been described in the state-of-the-art. Thus, proposing a set of energy-aware model transformations to apply on MoDisco models could certainly help energy oblivious developers to optimize and modernize, their programs. Furthermore, these model transformations could consider energy measurements modeled with SMM, as proposed in Section 3.3, to target the greediest methods in particular.

6.3.6 Improving monitoring of sensors and actuators networks

This last perspective focuses on Emit, our monitoring platform for sensors and actuators networks, and the meta-model used to defines the systems to be monitored. Considering the monitoring platform, a first perspective concerns the expressivity of the callback definition. Our current platform only enables the definition of simple callbacks, with limited guards (e.g., comparison of values, cast, storage in database). Future work could improve the callback definition to enable more analysis possibilities. Either by implementing more callbacks by hand, proposing a DSL that lets the user define the callbacks, or proposing an embedded coding environment in Emit enabling the definition of callbacks through GPLs. State-of-the-art monitoring platforms already propose this last option.

Another perspective concerns the verification and validation of CPSs. Emit could be used to perform dynamic verification of a running CPS, based on the messages sent and received by the devices deployed in the system. In opposition, static verification could be performed on sensors and actuators models, using model checking techniques, for instance.

A last perspective could be to improve the sensors and actuators meta-model to enable the definition of the devices behavior. As an example, it could rely on ThingML to define, through state machines, the behavior of sensors and actuators. ThingML would be finally used to generate the source code to deploy on the devices, thus leveraging MDE for all the aspects of the CPS.

List of Figures

List of Tables

Bibliography

[1] M. Acharya and B. Robinson. "Practical change impact analysis based on static program slicing for industrial software systems". In: *Proceedings of the 33rd international conference on software engineering*. ACM. 2011, pp. 746–755.

[2] K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia. "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*. CASCON '14. Markham, Ontario, Canada: IBM Corp., 2014, pp. 219–233.

[3] D. Akdur, V. Garousi, and O. Demirörs. "A survey on modeling and model-driven engineering practices in the embedded software industry". In: *Journal of Systems Architecture* 91 (2018), pp. 62 –82. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2018.09.007. URL: http://www.sciencedirect.com/science/article/pii/S1383762118302455.

[4] I. F. Akyildiz and I. H. Kasimoglu. "Wireless sensor and actor networks: research challenges". In: *Ad hoc networks* 2.4 (2004), pp. 351–367.

[5] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. "Wireless sensor networks: a survey". In: *Computer networks* 38.4 (2002), pp. 393–422.

[6] A. Alali, H. Kagdi, and J. I. Maletic. "What's a typical commit? A characterization of open source software repositories". In: *2008 16th IEEE International Conference on Program Comprehension*. IEEE. 2008, pp. 182–191.

[7] R. Ansorg and L. Schwabe. "Domain-specific modeling as a pragmatic approach to neuronal model descriptions". In: *International Conference on Brain Informatics*. Springer. 2010, pp. 168–179.

[8] M. W. Anwar, F. Azam, M. A. Khan, and W. H. Butt. "The Applications of Model Driven Architecture (MDA) in Wireless Sensor Networks (WSN): Techniques and Tools". In: *Future of Information and Communication Conference*. Springer. 2019, pp. 14–27.

[9] T. Apiwattanapong, A. Orso, and M. J. Harrold. "Efficient and precise dynamic impact analysis using execute-after sequences". In: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 432–441.

[10] D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu, et al. "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds". In: *2012 4th International Workshop on Modeling in Software Engineering (MISE)*. IEEE. 2012, pp. 50–56.

[11] R. B. Atitallah, S. Niar, A. Greiner, S. Meftali, and J. L. Dekeyser. "Estimating energy consumption for an MPSoC architectural exploration". In: *International Conference on Architecture of Computing Systems*. Springer. 2006, pp. 298–310.

[12] S. Azhar. "Building information modeling (BIM): Trends, benefits, risks, and challenges for the AEC industry". In: *Leadership and management in engineering* 11.3 (2011), pp. 241–252.

[13] R. Baheti and H. Gill. "Cyber-physical systems". In: *The impact of control technology* 12.1 (2011), pp. 161–166.

[14] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. "CodeSurfer/x86—A platform for analyzing x86 executables". In: *International Conference on Compiler Construction*. Springer. 2005, pp. 250–254.

[15] N. Bandener, C. Soltenborn, and G. Engels. "Extending DMM behavior specifications for visual execution and debugging". In: *International Conference on Software Language Engineering*. Springer. 2010, pp. 357–376.

[16] O. Barais, B. Combemale, and A. Wortmann. "Language Engineering with the GEMOC Studio". In: 2017.

[17] Y. Ben Maissa, F. Kordon, S. Mouline, and Y. Thierry-Mieg. "Modeling and Analyzing Wireless Sensor Networks with VeriSensor: An Integrated Workflow". In: *Transactions on Petri Nets and Other Models of Concurrency VIII*. Springer, 2013.

[18] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. "Neo4EMF, a scalable persistence layer for EMF models". In: *European Conference on Modelling Foundations and Applications*. Springer. 2014, pp. 230–241.

[19] L. Berardinelli, A. Di Marco, S. Pace, L. Pomante, and W. Tiberti. "Energy consumption analysis and design of energy-aware WSN agents in fUML". In: *European Conference on Modelling Foundations and Applications*. Springer. 2015, pp. 1–17.

[20] J. Bézivin. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188.

[21] J. Bézivin and O. Gerbé. "Towards a precise definition of the OMG/MDA framework". In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE. 2001, pp. 273–280.

[22] W. Binder, J. Hulaas, and P. Moret. "Advanced Java bytecode instrumentation". In: *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. 2007, pp. 135–144.

[23] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. "Regression test selection techniques: A survey". In: *Informatica* 35.3 (2011).

[24] M. Botts and A. Robin. "OpenGIS Sensor Model Language (SensorML) Implementation Specification (OGC 07–000)". In: (2007).

[25] A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier. "Powerapi: A software library to monitor the energy consumed at the process-level". In: *ERCIM News* 2013.92 (2013).

[26] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. "Execution framework of the gemoc studio (tool demo)". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 2016, pp. 84–89.

[27] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry. "Omniscient debugging for executable DSLs". In: *Journal of Systems and Software* 137 (2018), pp. 261–288.

[28] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry. "Advanced and efficient execution trace management for executable domain-specific modeling languages". In: *Software & Systems Modeling* (2017), pp. 1–37.

[29] E. Bousse, T. Mayerhofer, and M. Wimmer. "Domain-Level Debugging for Compiled DSLs with the GEMOC Studio (Tool Demo)". In: 2017.

[30] D. J. Brown and C. Reams. "Toward energy-efficient computing". In: *Communications of the ACM* 53.3 (2010), pp. 50–58.

[31] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. "Modisco: A model driven reverse engineering framework". In: *Information and Software Technology* 56.8 (2014), pp. 1012–1032. ISSN: 0950-5849.

[32] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. "MoDisco: a generic and extensible framework for model driven reverse engineering". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM. 2010, pp. 173–174.

[33] E. Bruneton. "ASM 3.0 A Java bytecode engineering library". In: *URL: http://download. forge. objectweb. org/asm/asmguide. pdf* (2007).

[34] B. Buck and J. K. Hollingsworth. "An API for runtime code patching". In: *The International Journal of High Performance Computing Applications* 14.4 (2000), pp. 317–329.

[35] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. "Ptolemy: A framework for simulating and prototyping heterogeneous systems". In: *Readings in hardware/software co-design*. 2001, pp. 527–543.

[36] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. "JRipples: A tool for program comprehension during incremental change". In: *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE. 2005, pp. 149–152.

[37] P. C. Canizares, A. Núñez, J. de Lara, and L. Llana. "MT-EA4Cloud: A Methodology For testing and optimising energy-aware cloud systems". In: *Journal of Systems and Software* 163 (2020), p. 110522.

[38] E. Capra, C. Francalanci, and S. A. Slaughter. "Measuring application software energy efficiency". In: *IT Professional* 14.2 (2012), pp. 54–61.

[39] A. Carette, M. Adel Ait Younes, G. Hecht, N. Moha, and R. Rouvoy. *Investigating the Energy Impact of Android Smells*. Tech. rep. 10. 2017. URL: https://hal.inria.fr/hal-01403485/file/carette-saner-17.pdf.

[40] E. Cariou, O. Le Goaer, L. Brunschwig, and F. Barbier. "A generic solution for weaving business code into executable models." In: *MODELS Workshops*. 2018, pp. 251–256.

[41] E. J. Chikofsky and J. H. Cross. "Reverse engineering and design recovery: A taxonomy". In: *IEEE software* 7.1 (1990), pp. 13–17.

[42] P. K. Chittimalli and M. J. Harrold. "Regression test selection on system requirements". In: *Proceedings of the 1st India software engineering conference*. ACM. 2008, pp. 87–96.

[43] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. "Process-level power estimation in vm-based systems". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–14.

[44] B. Combemale, X. Crégut, and M. Pantel. "A design pattern to build executable DSMLs and associated V&V tools". In: *2012 19th Asia-Pacific Software Engineering Conference*. Vol. 1. IEEE. 2012, pp. 282–287.

[45] G. Cook. "How clean is your cloud". In: *Catalysing an energy revolution* (2012), p. 11.

[46] L. Cruz, R. Abreu, and J.-N. Rouvignac. "Leafactor: Improving energy efficiency of android apps via automatic refactoring". In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2017, pp. 205–206.

[47] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. “Frama-c”. In: *International conference on software engineering and formal methods*. Springer. 2012, pp. 233–247.

[48] M. Dahm, J van Zyl, and E Haase. *The bytecode engineering library (BCEL)*. 2003.

[49] G. Daniel, G. Sunyé, and J. Cabot. “Scalable queries and model transformations with the mogwai tool”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2018, pp. 175–183.

[50] L. Daniel, M. Kojo, and M. Latvala. “Experimental evaluation of the CoAP, HTTP and SPDY transport services for Internet of Things”. In: *International Conference on Internet and Distributed Computing Systems*. Springer. 2014, pp. 111–123.

[51] P. Dantas, T. Rodrigues, T. Batista, F. C. Delicato, P. F. Pires, W. Li, and A. Y. Zomaya. “LWiSSy: A domain specific language to model wireless sensor and actuators network systems”. In: *4th International Workshop on Software Engineering for Sensor Network Applications*. IEEE. 2013, pp. 7–12.

[52] P. Daugherty, P. Banerjee, W. Negm, and A. E. Alter. “Driving unconventional growth through the industrial internet of things”. In: *Accenture Technology* (2015).

[53] J. De Lara and H. Vangheluwe. “AToM 3: A Tool for Multi-formalism and Meta-modelling”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2002, pp. 174–188.

[54] P. Derler, E. A. Lee, S. Tripakis, and M. Törngren. “Cyber-physical system design contracts”. In: *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. 2013, pp. 109–118.

[55] B. Dit, M. Wagner, S. Wen, W. Wang, M. Linares-Vásquez, D. Poshyvanyk, and H. Kagdi. “Impactminer: A tool for change impact analysis”. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 540–543.

[56] K. Doddapaneni, E. Ever, O. Gemikonakli, I. Malavolta, L. Mostarda, and H. Muccini. “A model-driven engineering framework for architecting and analysing wireless sensor networks”. In: *Proceedings of the 3rd International Workshop SESENA*. IEEE Press. 2012, pp. 1–7.

[57] B. Dougherty, J. White, and D. C. Schmidt. “Model-driven auto-scaling of green cloud computing infrastructure”. In: *Future Generation Computer Systems* 28.2 (2012), pp. 371–378.

[58] G. Dupe, M. Belaunde, R. Perruchon, H. Besnard, F. Guillard, and V. Oliveres. *SmartQVT*.

[59] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. "Xbase: implementing domain-specific languages for Java". In: *ACM SIGPLAN Notices* 48.3 (2012), pp. 112–121.

[60] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. "Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML". In: *International Conference on the Unified Modeling Language*. Springer. 2000, pp. 323–337.

[61] E. Engström and P. Runeson. "A qualitative survey of regression testing practices". In: *Product-Focused Software Process Improvement* (2010), pp. 3–16.

[62] E. Engström, P. Runeson, and M. Skoglund. "A systematic review on regression test selection techniques". In: *Information and Software Technology* 52.1 (2010), pp. 14–30.

[63] M. D. Ernst. "Static and dynamic analysis: Synergy and duality". In: *WODA 2003: ICSE Workshop on Dynamic Analysis*. 2003, pp. 24–27.

[64] M. Eysholdt and H. Behrens. "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*. 2010, pp. 307–309.

[65] J.-M. Favre. "Megamodeling and etymology-a story of words: From MED to MDE via MODEL in five milleniums". In: *In Dagstuhl Seminar on Transformation Techniques in Software Engineering, number 05161 in DROPS 04101. IFBI*. Citeseer. 2005.

[66] M. S. Feather. "A survey and classification of some program transformation approaches and techniques". In: *The IFIP TC2/WG 2.1 Working Conference on Program specification and transformation*. 1987, pp. 165–195.

[67] M. Follett and O. Hoeber. "ImpactViz: visualizing class dependencies and the impact of changes in software revisions". In: *Proceedings of the 5th international symposium on Software visualization*. 2010, pp. 209–210.

[68] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[69] A. German. "Software Static Code Analysis Lessons Learned©". In: *Crosstalk* (2003), pp. 13–17.

[70] M. Gligoric, L. Eloussi, and D. Marinov. “Practical regression test selection with dynamic file dependencies”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM. 2015, pp. 211–222.

[71] I. Gomes, P. Morgado, T. Gomes, and R. Moreira. “An overview on the static code analysis approach in software development”. In: *Faculdade de Engenharia da Universidade do Porto, Portugal* (2009).

[72] L. Gonnord and S. Mosser. “Practicing domain-specific languages: from code to models”. In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2018, pp. 106–113.

[73] A. Gosain and G. Sharma. “A Survey of Dynamic Program Analysis Techniques and Tools”. In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014: Volume 1*. Ed. by S. C. Satapathy, B. N. Biswal, S. K. Udgata, and J. Mandal. Cham: Springer International Publishing, 2015, pp. 113–122. ISBN: 978-3-319-11933-5. DOI: 10.1007/978-3-319-11933-5_13.

[74] G. Gousios, M. Pinzger, and A. v. Deursen. “An exploratory study of the pull-based software development model”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 345–355.

[75] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. Cavalli. “Detecting control flow in smarphones: Combining static and dynamic analyses”. In: *Cyberspace Safety and Security*. Springer, 2012, pp. 33–47.

[76] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. “An empirical study of regression test selection techniques”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2001).

[77] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. “Static analysis of energy consumption for LLVM IR programs”. In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. 2015, pp. 12–21.

[78] N. Guarino, C. Welty, et al. “Towards a methodology for ontology-based model engineering”. In: *Proceedings of the ECOOP-2000 Workshop on Model Engineering*. 2000.

[79] J. Han and M. Orshansky. “Approximate computing: An emerging paradigm for energy-efficient design”. In: *2013 18th IEEE European Test Symposium (ETS)*. IEEE. 2013, pp. 1–6.

[80] S. Hao, D. Li, W. G. Halfond, and R. Govindan. “Estimating mobile application energy consumption using program analysis”. In: *2013 35th international conference on software engineering (ICSE)*. IEEE. 2013, pp. 92–101.

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

[82] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa. "ThingML: A Language and Code Generation Framework for Heterogeneous Targets". In: *19th ACM/IEEE International Conference MoDELS*. ACM, 2016.

[83] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. "Energy profiles of java collections classes". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 225–236.

[84] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damásio. "On the precision and accuracy of impact analysis techniques". In: *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*. IEEE. 2008, pp. 513–518.

[85] A. Hegedus, G. Bergmann, I. Ráth, and D. Varró. "Back-annotation of simulation traces with change-driven model transformations". In: *Software Engineering and Formal Methods, 2010*.

[86] Á. Hegedüs, I. Ráth, and D. Varró. "Replaying execution trace models for dynamic modeling languages". In: *Periodica Polytechnica Electrical Engineering and Computer Science* 56.3 (2012), pp. 71–82.

[87] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. "Greenminer: A hardware based mining software repositories software energy consumption framework". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. 2014, pp. 12–21.

[88] J. M. Hirst, J. R. Miller, B. A. Kaplan, and D. D. Reed. *Watts up? pro ac power meter for automated energy recording*. 2013.

[89] U. Hunkeler, H. L. Truong, and A. Stanford-Clark. "MQTT-SA publish/-subscribe protocol for Wireless Sensor Networks". In: *3rd International Conference on Communication Systems Software and Middleware and Workshops*. IEEE. 2008.

[90] J. Hutchinson, J. Whittle, and M. Rouncefield. "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure". In: *Science of Computer Programming* 89 (2014), pp. 144–161.

[91] D. Jackson and M. Rinard. "Software analysis: A roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, pp. 133–145.

[92] K. Janowicz, A. Haller, S. J. Cox, D. Le Phuoc, and M. Lefrançois. "SOSA: A lightweight ontology for sensors, observations, samples, and actuators". In: *Journal of Web Semantics* 56 (2019), pp. 1–10.

[93] *JArchitect : Java Static Analysis and Code Quality Tool*. (accessed July 8, 2020). URL: https://www.jarchitect.com/.

[94] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. "Mashup of metalanguages and its implementation in the kermeta language workbench". In: *Software & Systems Modeling* 14.2 (2015), pp. 905–920.

[95] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. "ATL: A model transformation tool". In: *Science of computer programming* 72.1-2 (2008), pp. 31–39.

[96] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. "ATL: a QVT-like transformation language". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006, pp. 719–720.

[97] S. Kent. "Model driven engineering". In: *International Conference on Integrated Formal Methods*. Springer. 2002, pp. 286–298.

[98] D. Kim, J.-E. Hong, I. Yoon, and S.-H. Lee. "Code refactoring techniques for reducing energy consumption in embedded computing environment". In: *Cluster Computing* 21.1 (2018), pp. 1079–1095.

[99] K.-D. Kim and P. R. Kumar. "Cyber–physical systems: A perspective at the centennial". In: *Proceedings of the IEEE* 100.Special Centennial Issue (2012), pp. 1287–1308.

[100] Z. King, M. Sayagh, and A. Hindle. "Energy Profiles of Java Collections Classes". In: (2016).

[101] D. S. Kolovos, R. F. Paige, and F. A. Polack. "The epsilon transformation language". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 46–60.

[102] S. Kubler, K. Främling, and A. Buda. "A standardized approach to deal with firewall and mobility policies in the IoT". In: *Pervasive and Mobile Computing* 20 (2015).

[103] T. Kurpick, C. Pinkernell, M. Look, and B. Rumpe. "Modeling cyber-physical systems: model-driven specification of energy efficient buildings". In: *Proceedings of the Modelling of the Physical World Workshop*. 2012, pp. 1–6.

[104] Y.-W. Kwon and E. Tilevich. "Reducing the energy consumption of mobile applications behind the scenes". In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 170–179.

[105] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. "Pebil: Efficient static binary instrumentation for linux". In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE. 2010, pp. 175–183.

[106] J. Law and G. Rothermel. "Whole program path-based dynamic impact analysis". In: *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society. 2003.

[107] E. Le Sueur and G. Heiser. "Dynamic voltage and frequency scaling: The laws of diminishing returns". In: *Proceedings of the 2010 international conference on Power aware computing and systems*. 2010, pp. 1–8.

[108] S.-W. Lee and J.-L. Gaudiot. "Throttling-based resource management in high performance multithreaded architectures". In: *IEEE Transactions on Computers* 55.9 (2006), pp. 1142–1152.

[109] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. "An extensive study of static regression test selection in modern software evolution". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 583–594.

[110] E. Lepore and B. Loewer. "Translational semantics". In: *Synthese* (1981), pp. 121–133.

[111] B. Li, X. Sun, H. Leung, and S. Zhang. "A survey of code-based change impact analysis techniques". In: *Software Testing, Verification and Reliability* 23.8 (2013), pp. 613–646.

[112] D. Li, S. Hao, J. Gui, and W. G. Halfond. "An empirical study of the energy consumption of android applications". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 121–130.

[113] D. Li, S. Hao, W. G. Halfond, and R. Govindan. "Calculating source line level energy information for android applications". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 2013, pp. 78–89.

[114] X. Li, P. J. Ortiz, J. Browne, D. Franklin, J. Y. Oliver, R. Geyer, Y. Zhou, and F. T. Chong. "Smartphone evolution and reuse: Establishing a more sustainable model". In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE. 2010, pp. 476–484.

[115] R. A. Light. "Mosquitto: server and client implementation of the MQTT protocol". In: *The Journal of Open Source Software* 2.13 (2017), p. 265.

[116] K. Liu, G. Pinto, and Y. D. Liu. "Data-oriented characterization of application-level energy optimization". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2015, pp. 316–331.

[117] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices* 40.6 (2005), pp. 190–200.

[118] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. "An empirical analysis of flaky tests". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 643–653.

[119] I. Manotas, L. Pollock, and J. Clause. "SEEDS: a software engineer's energy-optimization decision support framework". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 503–514.

[120] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. "DiSL: a domain-specific language for bytecode instrumentation". In: *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. 2012, pp. 239–250.

[121] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. "xMOF: Executable DSMLs based on fUML". In: *International Conference on Software Language Engineering*. Springer. 2013, pp. 56–75.

[122] T. Mayerhofer, M. Wimmer, L. Burgueño, and A. Vallecillo. *Specifying quantities in software models*. Tech. rep. Submitted, 2018.

[123] Q. Medini. *IKV++ technologies home*. 2011.

[124] T. Mens and P. Van Gorp. "A taxonomy of model transformation". In: *Electronic notes in theoretical computer science* 152 (2006), pp. 125–142.

[125] S. Mittal. "A survey of techniques for improving energy efficiency in embedded computing systems". In: *International Journal of Computer Aided Engineering and Technology* 6.4 (2014), pp. 440–459.

[126] M. Monperrus, A. Beugnard, and J. Champeau. "A definition of" abstraction level" for metamodels". In: *2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. IEEE. 2009, pp. 315–320.

[127] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. "On the interplay between software testing and evolution and its effect on program comprehension". In: *Software evolution*. Springer, 2008, pp. 173–202.

[128] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. "Earmo: An energy-aware refactoring approach for mobile apps". In: *IEEE Transactions on Software Engineering* 44.12 (2017), pp. 1176–1206.

[129] D. Mosteller, M. Haustermann, D. Moldt, and D. Schmitz. "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, 2019, pp. 101–125.

[130] J. Muñoz, P. Valderas, V. Pelechano, and O. Pastor. "Requirements engineering for pervasive systems. a transformational approach". In: *14th IEEE International Requirements Engineering Conference (RE'06)*. IEEE. 2006, pp. 351–352.

[131] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. "Acceleo user guide". In: *See also http://acceleo.org/doc/obeo/en/acceleo-2.6-user-guide. pdf* 2 (2006), p. 157.

[132] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. "Applying static analysis to software architectures". In: *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. 1997, pp. 77–93.

[133] N. Nethercote. *Dynamic binary analysis and instrumentation*. Tech. rep. University of Cambridge, Computer Laboratory, 2004.

[134] H. Neuhaus and M. Compton. "The semantic sensor network ontology". In: *AGILE workshop on challenges in geospatial data harmonisation, Hannover, Germany*. 2009, pp. 1–33.

[135] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. "A preliminary study of the impact of software engineering on GreenIT". In: *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*. 2012, pp. 21–27. ISBN: 9781467318327. DOI: 10.1109/GREENS.2012.6224251. URL: https://hal.inria.fr/hal-00681560v1.

[136] A. Noureddine, R. Rouvoy, and L. Seinturier. "A review of energy measurement approaches". In: *ACM SIGOPS Operating Systems Review* 47.3 (2013), pp. 42–49.

[137] A. Noureddine, R. Rouvoy, and L. Seinturier. "Unit testing of energy consumption of software libraries". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM. 2014, pp. 1200–1205.

[138] O. M. G. Object Management Group. *Structured Metrics Metamodel*. URL: https://www.omg.org/spec/SMM/1.2/.

[139] M. F. Oliveira, E. W. Brião, F. A. Nascimento, and F. R. Wagner. "Model driven engineering for MPSOC design space exploration". In: *Proceedings of the 20th annual conference on Integrated circuits and systems design*. 2007, pp. 81–86.

[140] M. d. S. Oliveira, L. B. de Brisolara, L. Carro, and F. R. Wagner. "Early embedded software design space exploration using UML-based estimation". In: *Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*. IEEE. 2006, pp. 24–32.

[141] A. Orso, N. Shi, and M. J. Harrold. "Scaling regression testing to large software systems". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 29. 6. ACM. 2004, pp. 241–251.

[142] P. J. Ortiz, J. Browne, D. Franklin, J. Y. Oliver, R. Geyer, Y. Zhou, and F. T. Chong. "Smartphone Evolution and Reuse : Establishing a More Sustainable Model Smartphone Evolution and Reuse : Establishing a more Sustainable Model". In: 90 (2015). DOI: 10.1109/ICPPW.2010.70.

[143] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. "Morsa: A scalable approach for persisting and accessing large models". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2011, pp. 77–92.

[144] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. "What Do Programmers Know about Software Energy Consumption?" In: *IEEE Software* 33.3 (2016), pp. 83–89. ISSN: 07407459. DOI: 10.1109/MS.2015.83.

[145] M. Van de Panne and E. Fiume. "Sensor-actuator networks". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993.

[146] H. Partsch and R. Steinbrüggen. "Program transformation systems". In: *ACM Computing Surveys (CSUR)* 15.3 (1983), pp. 199–236.

[147] D Pavithra and R. Balakrishnan. "IoT based monitoring and control system for home automation". In: *2015 global conference on communication technologies (GCCT)*. IEEE. 2015, pp. 169–173.

[148] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. "SPOON: A library for implementing analyses and transformations of Java source code". In: *Software - Practice and Experience* 46.9 (2016), pp. 1155–1179. ISSN: 1097024X. DOI: 10.1002/spe.2346. arXiv: 1008.1900.

[149] R. Pereira, M. Couto, J. Saraiva, J. Cunha, and J. P. Fernandes. "The influence of the Java collection framework on overall energy consumption". In: *GREENS'16*. ACM. 2016.

[150] R. Pereira, J. Saraiva, H. I. Tec, N. Lincs, and J. P. Fernandes. "The Influence of the Java Collection Framework on Overall". In: (2016).

[151] G. Pinto and F. Castor. "Energy efficiency: a new concern for application software developers". In: *Communications of the ACM* 60.12 (2017), pp. 68–75.

[152] G. Pinto, F. Castor, and Y. D. Liu. "Mining questions about software energy consumption Understanding Open-Source Software and Communities View project Transfering knowledge to software engineering practice View project Mining Questions about Software Energy Consumption". In: (). DOI: 10.1145/2597073.2597110. URL: www.stackoverflow.com/questions/413227.

[153] G. Pinto, F. Castor, and Y. D. Liu. "Understanding energy behaviors of thread management constructs". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 2014, pp. 345–360.

[154] G. Pinto, F. Soares-Neto, and F. Castor. "Refactoring for energy efficiency: a reflection on the state of the art". In: *Proceedings of the Fourth International Workshop on Green and Sustainable Software*. IEEE Press. 2015, pp. 29–35.

[155] G. D. Plotkin. "A structural approach to operational semantics". In: (1981).

[156] R. Pohjonen, J.-P. Tolvanen, and M Consulting. "Automated production of family members: Lessons learned". In: *Proc. of PLEES* 2 (2002), pp. 49–57.

[157] R. Priego, A. Armentia, E. Estévez, and M. Marcos. "Modeling techniques as applied to generating tool-independent automation projects". In: *at-Automatisierungstechnik* 64.4 (2016), pp. 325–340.

[158] G. Procaccianti, H. Fernández, and P. Lago. "Empirical evaluation of two best practices for energy-efficient software development". In: *Journal of Systems and Software* 117 (2016), pp. 185–198. ISSN: 01641212. DOI: 10.1016/j.jss.2016.02.035. URL: https://wiki.cs.vu.nl/green{_}software/index.php/Best{_}practices{_}for{_}.

[159] V. P. Ranganath and J. Hatcliff. "Slicing concurrent Java programs using Indus and Kaveri". In: *International Journal on Software Tools for Technology Transfer* 9.5-6 (2007), pp. 489–504.

[160] C. Reams. "Toward Efficient Computing". In: ().

[161] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. "Chianti: a tool for change impact analysis of java programs". In: 39.10 (2004), pp. 432–448.

[162] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. "Chianti: a tool for change impact analysis of java programs". In: *ACM Sigplan Notices*. Vol. 39. 10. ACM. 2004, pp. 432–448.

[163] F. Rieger and C. Bockisch. "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*. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 125–128. ISBN: 9781450368667. DOI: 10.1145/3341105.3374105. URL: https://doi.org/10.1145/3341105.3374105.

[164] F. Rieger and C. Bockisch. "Survey of approaches for assessing software energy consumption". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*. 2017, pp. 19–24.

[165] S. a. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. "JouleSort: a balanced energy-efficiency benchmark". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, pp. 365–376.

[166] J. Rocheteau, V. Gaillard, and L. Belhaj. "How Green Are Java Best Coding Practices?." In: *SMARTGREENS*. 2014, pp. 235–246.

[167] T. Rodrigues, P. Dantas, P. F. Pires, L. Pirmez, T. Batista, C. Miceli, and A. Zomaya. "Model-driven development of wireless sensor network applications". In: *IFIP 9th International Conference on Embedded and Ubiquitous Computing*. IEEE. 2011.

[168] G. Rothermel and M. J. Harrold. "Analyzing regression test selection techniques". In: *IEEE Transactions on software engineering* 22.8 (1996), pp. 529–551.

[169] J. Rumbaugh, I. Jacobson, and G. Booch. "The unified modeling language". In: *Reference manual* (1999).

[170] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. "Initial explorations on design pattern energy usage". In: *2012 First International Workshop on Green and Sustainable Software (GREENS)*. IEEE. 2012, pp. 55–61.

[171] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. "EnerJ: Approximate data types for safe and general low-power computation". In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 164–174.

[172] E. Saxe and S. Microsystems. "Power-Efficient Software". In: (), pp. 1–8.

[173] J. Scaramella. “Solutions for the Datacenter ’ s Thermal Challenges”. In: January (2007).

[174] M. J. Scaramella and M. Eastwood. “Solutions for the datacenter’s thermal challenges”. In: *IDC, January* (2007).

[175] G. Sedrakyan and M. Snoeck. “Enriching Model Execution with Feedback to Support Testing of Semantic Conformance between Models and Requirements”. In: (2016).

[176] B. Selic. “The pragmatics of model-driven development”. In: *IEEE software* 20.5 (2003), pp. 19–25.

[177] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner. “United states data center energy usage report”. In: (2016).

[178] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner. *United States Data Center Energy Usage Report*. Tech. rep. 2016.

[179] W. G. da Silva, L. Brisolara, U. B. Correa, and L. Carro. “Evaluation of the impact of code refactoring on embedded software efficiency”. In: *Proceedings of the 1st Workshop de Sistemas Embarcados*. 2010, pp. 145–150.

[180] D. Singh and W. J. Kaiser. “The atom LEAP platform for energy-efficient embedded computing”. In: (2010).

[181] K. J. Singh and D. S. Kapoor. “Create Your Own Internet of Things: A survey of IoT platforms.” In: *IEEE Consumer Electronics Magazine* 6.2 (2017), pp. 57–68.

[182] J. Stasko and E. Zhang. “Focus+ context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations”. In: *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings*. IEEE. 2000, pp. 57–65.

[183] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[184] M. łgorzata Steinder and A. S. Sethi. “A survey of fault localization techniques in computer networks”. In: *Science of computer programming* 53.2 (2004), pp. 165–194.

[185] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier. “Formalizing execution semantics of UML profiles with fUML models”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2014, pp. 133–148.

[186] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan. "Performance evaluation of MQTT and CoAP via a common middleware". In: *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. IEEE. 2014, pp. 1–6.

[187] C. Thompson, J. White, B. Dougherty, and D. C. Schmidt. "Optimizing mobile application performance with model–driven engineering". In: *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer. 2009, pp. 36–46.

[188] M. M. Tikir and J. K. Hollingsworth. "Efficient instrumentation for code coverage testing". In: *ACM SIGSOFT Software Engineering Notes* 27.4 (2002), pp. 86–96.

[189] C. Trabelsi, R. Ben Atitallah, S. Meftali, J.-L. Dekeyser, and A. Jemai. "A model-driven approach for hybrid power estimation in embedded systems design". In: *EURASIP Journal on Embedded Systems* 2011 (2011), pp. 1–15.

[190] *Unravel*. (accessed July 8, 2020). URL: https://www.nist.gov/itl/ssd/unravel-project/.

[191] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. "Soot: A Java bytecode optimization framework". In: *CASCON First Decade High Impact Papers*. 2010, pp. 214–224.

[192] A. Van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A framework for application performance monitoring and dynamic software analysis". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. 2012, pp. 247–248.

[193] G. Venkatesh. "Experimental results from dynamic slicing of C programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.2 (1995), pp. 197–216.

[194] C. Vidal, C. Fernández-Sánchez, J. Díaz, and J. Pérez. "A model-driven engineering process for autonomic sensor-actuator networks". In: *International Journal of Distributed Sensor Networks* 2015 (2015), p. 18.

[195] E. Visser. "A survey of rewriting strategies in program transformation systems". In: *Electronic Notes in Theoretical Computer Science* 57.2 (2001).

[196] V. Viyović, M. Maksimović, and B. Perisić. "Sirius: A rapid development of DSM graphical editor". In: *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE. 2014, pp. 233–238.

[197] T. Wang and A. Roychoudhury. "Dynamic slicing on Java bytecode traces". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.2 (2008), pp. 1–49.

[198] M. Webb et al. “Smart 2020: Enabling the low carbon economy in the information age”. In: *The Climate Group. London* 1.1 (2008), pp. 1–1.

[199] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. “Industrial adoption of model-driven engineering: Are the tools really the problem?” In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2013, pp. 1–17.

[200] B. A. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, and D. Marsh. “Industrial perspective on static analysis”. In: *Software Engineering Journal* 10.2 (1995), pp. 69–75.

[201] R. Winterhalter. *Byte Buddy*.

[202] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. “A survey on software fault localization”. In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

[203] Q. Wu, M. Pedram, and X. Wu. “Clock-gating and its application to low power design of sequential circuits”. In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 47.3 (2000), pp. 415–420.

[204] B. M. Yildiz, C. Bockisch, A. Rensink, and M. Aksit. “A Java Bytecode Metamodel for Composable Program Analyses”. In: *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer. 2017, pp. 30–40.

[205] S. Yoo and M. Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

[206] H. Zhang and H Hoffman. “A quantitative evaluation of the RAPL power control system”. In: *Feedback Computing* (2015).

[207] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. “Accurate online power estimation and automatic battery behavior based power model generation for smartphones”. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 2010, pp. 105–114.

[208] L. Zhang. “Hybrid regression test selection”. In: *Proceedings of the 40th International Conference on Software Engineering*. ACM. 2018, pp. 199–209.

[209] L. Zhang, M. Kim, and S. Khurshid. “FaultTracer: a change impact and regression fault analysis tool for evolving Java programs”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Méthodes dirigées par les modèles pour l'analyse dynamique appliquées a l'ingénierie de logiciels verts

Mots-clés : Ingénierie dirigée par les modèles, analyse dynamique, systèmes cyber-physique, estimation énergétique, selection de tests de regression.

Résumé : L'ingénierie dirigée par les *modèles* est un processus de développement qui centralise l'utilisation de modèles à toutes les étapes de la création d'applications. Lors de la phase de conception d'une application, il est commun d'analyser son *modèle* afin de vérifier sa conformité. L'analyse statique de modèle est courante, cependant le manque d'informations dynamiques dans les modèles freine la détection d'anomalies tôt dans le cycle de développement. La détection d'anomalies de consommation énergétique tôt dans le cycle de développement est importante, et nécessite d'analyser dynamiquement le modèle. Cette thèse présente deux approches permettant l'analyse dynamiques de modèles. Une première contribution injectes des traces d'exécution au sein de modèles de code source, et une seconde contribution génère une application de surveillance de système cyber-physique, à partir de son modèle de conception. Plusieurs analyses dynamiques sont effectués en se reposant sur ces approches, notamment dans le cadre de l'efficacité énergétique et de l'optimisation des tests de non regression.

Model-driven Methods for Dynamic Analysis applied to Energy-Aware Software Engineering

Keywords : Model-Driven Engineering, Dynamic Analysis, Cyber-Physical Systems, Energy Efficiency, Regression Test Selection

Abstract: Model-Driven Engineering (MDE) is a process that promotes models as the central key element for all phases in a software development lifecycle. Improving the quality of a software at design time can be done by performing analysis on the model it is designed with. Performing static analysis on models is extremely common during development phases, however the limited possibilities of dynamic analysis in models prevents early improvements of software and system. This lack of dynamic analysis options is especially important in the context of energy aware software engineering: good design choices must be done early in the development cycle to optimize the energy consumption. In this thesis we propose several approaches for performing dynamic analysis on models. A first contribution injects execution traces into source code model, and a second one generates monitoring application of cyber-physical system based on design model. Several dynamic analysis use-cases for energy-efficiency are presented: either for energy estimation or to lighten the cost of regression testing.