Exploiting the Internet of Things to Teach Domain-Specific Languages and Modeling

The ArduinoML project

Sébastien Mosser, Philippe Collet, and Mireille Blay-Fornarino

Univ. Nice Sophia Antipolis, I3S, UMR 7271, 06900 Sophia Antipolis, France
{mosser,collet,blay}@i3s.unice.fr

Abstract. The Computer Science department of the University of Nice – Sophia Antipolis is offering a course dedicated to *Model-Driven Engineering* (MDE) in its graduate curriculum. This course exists since 2006, and was usually badly perceived by students, despite many reorganizations of both course contents and teaching methods. This paper is an experience report that describes the latest version of this course. It relies on a case study leveraging domain-specific languages and open-source microcontrollers to support the domain modeling of Internet of Things pieces of software. It exploits domain modeling as a pivot to illustrate MDE concepts (*e.g.*, meta-modeling, model transformation), coupled to very practical labs where students experiment their models on real microcontrollers. This new version was well received by students.

1 Introduction

Our community is aware of issues related to the teaching of modeling. Difficulties with the abstraction and hindsight needed in metamodeling, or with the *heaviness* of the available tools such as the Eclipse Modeling Framework (EMF) [7], are typical examples. Such issues make it very challenging to transfer *Model-Driven Engineering* (MDE) to students. To tackle this challenge, Batory *et al* reported an experiment made at the University of Texas at Austin that uses database concepts as a technological background to teach MDE [1]. This experiment is decisive as it makes explicit that it is possible to teach MDE without introducing the complexity of tools such as the EMF. However, it uses *classical* MDE examples (*e.g.*, class-based structural meta-models, finite state machines), with trivial applications such as Java code generation.

We tried in 2008 to introduce in our course contents a set of very similar examples [2]. In addition to the complexity of exploiting the EMF (as spotted by Batory *et al*), students pointed out the artificial dimension of such examples. Page and Rose published a tribune entitled "*Lies, Damned Lies and UML2Java*" in the Journal of Object Technology, stating that "*It would be much more interesting to read about MDE scenarios that don't involve the infamous UML2Java transformation*" [6]. Based on this assumption we introduced in 2012 a case study based on the Internet of Things, where students had to model sensor

dashboards. We exploited the Sensapp platform [5], and students were asked to provide a meta-model and the associated tooling to plug data collected from sensors by Sensapp into graphical widgets (targeting HTML and Javascript implementations). In the yearly feedback, students were still complaining about the EMF, but appreciated the theme. They emphasized the practical dimension of the assignment, and stated that seeing a model being transformed into a graphical dashboard was more enjoyable that into plain Java code. However, the main negative feedback was the same: they did not see the real benefit of using models in this context. The use case was still too trivial, and it was simpler for them to program a dashboard in plain Javascript than to follow an MDE approach.

As a consequence, we proposed in Fall 2013 a new version of this course. Our objectives were threefold: *(i)* to emphasize the benefits of using models by exploiting a non-trivial and realistic case study (O_1), *(ii)* to support practical lab sessions associated to the course (O_2) and *(iii)* to provide a project description abstract enough to decouple it from tools (O_3). The remainder of this paper is organized as follows: Sec. 2 describes the case study used in this new version of the course, Sec. 3 provides a non-exhaustive description of the concepts that can be illustrated with this case study, and finally Sec. 4 concludes this experience report by discussing student feedback and sketching several perspectives to be implemented during the Fall semester 2014.

2 The ArduinoML Case Study

Based on our experience and industrial contacts, we decided to focus the course contents on *Domain-Specific Language*s (DSLs). Whittle *et al* reported in a industrial survey about MDE practices that "*There's also widespread use of mini-DSLs, even within a single project*" [8], strengthening this assumption. This concept acts as a common theme for an 8 weeks course (4 hours per week) followed by 39 graduate students in 2013. To illustrate it, and considering the success of the case study based on the Internet of Things used in 2012, we leveraged research work made by Fleurey *et al* [3] to create the ArduinoML case study, *i.e.*, the definition of a modeling language for the Arduino Uno[1] micro-controller platform.

2.1 Project Description

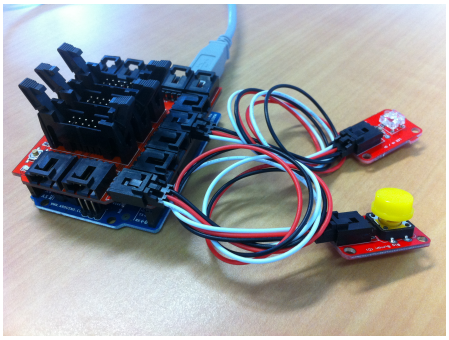
The project uses as its initial assumption the existing gap between *(i)* a piece of software implemented on top of the Internet of Things and *(ii)* the initial intention of the user. We consider as an illustrating use case in the project description one of the tutorials defined on the Arduino website[2]:

> ***Switch****: Alice owns a button (a sensor) and a LED (an actuator). She wants to connect these two things together in order to switch on the LED*

[1] `http://arduino.cc/en/Main/ArduinoBoardUno`
[2] `http://www.arduino.cc/en/Tutorial/Switch`

```
1  int state = LOW; int prev = HIGH;
2  long t = 0; long debounce = 200;
3  void setup() {
4    pinMode(8,INPUT);
5    pinMode(11, OUTPUT);
6  }
7  void loop() {
8   int reading = digitalRead(8);
9   if (reading == HIGH && prev == LOW
10        && millis() - t > debounce) {
11     if (state == HIGH) {
12       state = LOW;
13     } else { state = HIGH; }
14     time = millis();
15   }
16   digitalWrite(11, state);
17   prev = reading;
18 }
```

(a) Executable code (from tutorial)



(b) Hardware (Electronic Bricks)

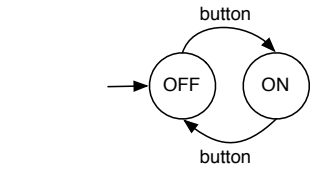
Fig. 1: Implementing the "*Switch*" use case.

> *when the button is pushed, and switch it off when the button is pressed again.*

At the hardware level (Fig. 1b), it means to connect the button and the LED to different pins of an Arduino Uno platform, and then flash the piece of code described in Fig. 1a on the micro-controller. This code starts by defining a set of global variables to store the state of the LED and handle signal noise (debounce mechanism). Then, it declares a `setup` function, called once when the micro-controller is started. This function states that the "thing" plugged into the pin numbered 8 is an input (*i.e.*, the button), and that the one plugged into the pin numbered 11 is an output (*i.e.*, the LED). Finally, it declares a `loop` function that is continuously executed by the micro-controller. This function implements the behavior of the "*Switch*" use case by switching on or off the LED based on the status of the button. It takes into account the electronic noise that exists at the I/O level by implementing the debouncing mechanism (line 10 & 14).

Clearly, even if this code looks like a correct implementation, it is not expressing Alice's initial intention in her own terms. Even this toy example demonstrates the gap that exists between what the user wanted to do with her things, and how the concrete implementation on top of an Arduino micro-controller differs in terms of concepts and abstractions.

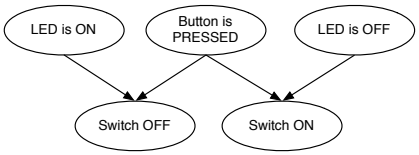
To perform the assignment associated to this project, students form pairs and work together to create a DSL supporting users who want to properly exploits things (sensors and actuators), at the right level of abstraction. They must demonstrate their language on three use cases: *(i)* the "*Switch*" one, *(ii)* a "*Level Crossing*" scenario involving a barrier, a set of traffic lights coupled to an alarm and an emergency button for imminent danger such as a pedestrian who had fall



(a) State Machine

Button	⇑	⇑	⇓	⇓
LED	ON	OFF	ON	OFF
switch	ON	OFF	OFF	ON

(b) Decision Table



(c) Dependency Network

when $button \wedge led$ do $switch(led, off)$

when $button \wedge \neg led$ do $switch(led, on)$

(d) Business Rules

Fig. 2: Modeling the "*Switch*" use case.

on the tracks, and finally *(iii)* imagine their very own use case, exploiting the different sensors and actuators available in the lab.

2.2 Relevance to the Targeted Objectives

The main objective of the ArduinoML case study is to emphasize the benefits of model-driven approaches (O_1). With the "*Switch*" use case, students had immediately spotted the difference between low-level programming (which may be tricky, *e.g.*, handling circuit noise) at the Arduino level, as opposed to the initial system one can sketch on paper. Contrarily to approaches that target abstract business domains and high-level languages such as Java or Javascript, forcing the student to work with a physically constrained domain (*e.g.*, 32Kb of memory, no high level language constructions) clearly emphasize the benefits of exploiting models at the end-user level. In this context, an end user cannot work at the code level to verify the behavior of a system, or even simply discuss about it with someone else: such a representation does not support users' understanding. Another interesting property of the project is that even if the underlying meta-model is not a classical one (*e.g.*, library, coffee vending machine or classes-to-relational transformation examples), it can still be exposed to the end-user using classical DSL families [4], *e.g.*, state machines (Fig. 2a), decision table (Fig. 2b), dependency network (Fig. 2c) or business rules (Fig. 2d).

To strengthen the practical implementation (O_2), we relied on an hardware-based approach by providing to each student groups a micro-controller and a set of sensors and actuators. A complete package costs around $70, including a micro-controller, 10 sensors and actuators and the necessary wiring. These packages can be easily shared by different groups, as Autodesk is providing an impressive simulator available in a web browser[3] for day-to-day work. Typically,

[3] `http://123d.circuits.io/`

students were using the simulated environment at home, and used the real hardware in shared mode during the lab sessions (4 hours, once a week).

Finally, the last objective is achieved by describing the case study as a call for bids. As a consequence, the project description is completely independent of any technological background (O_3). The project description solely describes the problem and gives evaluation criteria (see next section) to be used to rank the proposed solutions. We list in the Appendix section the different technologies used in Fall 2013 to support the implementation of this project.

2.3 Project Implementation & Evaluation

The project is divided into two phases, following a lean approach. During the two first weeks, students have to develop the kernel of their language, *i.e.*, a meta-model to represent what the user wants to do with her things and a preliminary code generator targeting the Arduino platform. Following a meta-model first approach supports the student while capturing the business domain they are working with. This milestone is validated on top of the initial "*Switch*" use case described in the previous section. One of the lab sessions is supervised by a colleague who acts as a simple user, pretending not to understand any sentence that is too technical. Thus, students have to use only the domain vocabulary (*e.g.*, button, switching on the light) while discussing with her. Students deliver their implementation and a short report describing it. They receive feedback within a week about their design decisions. As a consequence of such a lean approach, they deliver a running prototype of their language as early as possible, and get feedback to improve their design within a reasonable amount of time. This early milestone also allows the teaching staff to identify errors such as a wrong level of abstraction in the meta-model (*e.g.*, a "Button" concept), or a bad structural representation that will prevent the generation of running artifacts.

Then, students exploit the remaining six weeks to *(i)* improve their meta-model based on the received feedback, *(ii)* design a concrete syntax associated to this meta-model and finally *(iii)* extend their project to support new requirements. The call for bids describes sixteen extensions available on top of their kernel, each one associated to a cost (from 1 to 4 stars). The final delivery must include at least 5 stars in addition to the kernel. Introducing this variability in the project description was a real success in 2013, as students could choose extensions related to their other courses. For example, students involved in HCI courses mainly took extensions related to graphical representations of sensed data, or the implementation of ergonomic constraints on top of the inputs and outputs of the modeled application. Students more interested in software architecture chose extensions related to the communication of multiple models to support interacting systems, or even implement a model composition operator to support the deployment of multiple models on the very same piece of hardware. The different extensions are listed in Tab. 1 and discussed in the next session, with respect to the modeling concepts they illustrate.

The evaluation of the project is based on multiple criteria. The first and more important criterion is the expressiveness of the designed language with

respect to the different use cases expressed in the call for bids. To strengthen this evaluation, evaluators also use an undocumented use case, and uses the designed language to model a system that is not described in the call for bids. Other criteria includes meta-model correctness, concrete syntax expressiveness, code generator implementation, quality of examples used to assess the extensions and global elegance of the designed system. For the code generation part, the size of the generated program is also taken into account as a penalty when not reasonable, as Arduino micro-controller only contains 32Kb of memory.

Table 1: Extensions available to enhance the ArduinoML kernel.

Id	Extension	Cost	Description
1	Specifying execution frequency	1	Instead of having the program running in continuous mode, provide a way to specify an execution frequency used to halt the program and save energy.
2	Hardware kit descriptor	1	Provide a way to model the hardware (sensors and actuators) available in each kit provided by different manufacturers, and guide the user while choosing elements.
3	Remote communication	1	Model the communication between multiple systems to support the communication of different micro-controllers.
4	Exception throwing	1	Introduce in the DSL an exception mechanism to identify errors.
5	Support analogical sensors	2	Enhance the expressiveness to support behaviors that handle analogical sensors ($v \in [0, 1024[$) in addition to digital sensors ($v \in \{HIGH, LOW\}$).
6	Pin allocation generator	2	Based on the description of the user, produce the blueprint that explains how the different sensors and actuators must be connected to the micro-controller.
7	Ergonomic constraints	2	Enhance the DSL to support constraints associated to the targeted users. For example, visually impaired people will not be receptive to LED, and a buzzer must be used as a replacement.
8	Morse code generator	2	Define a new DSL that supports the definition of code such as the Morse alphabet. This DSL will be then compiled to ArduinoML, used as an execution platform.
9	Querying	3	Provide a way to query the modeled systems, such as "which elements are impacted when I push this button?", or "Is there any holes in my model, *i.e.*, combinations of sensors values that are not related to any actuators?"
10	System composition	3	Consider two systems s (the "*Switch*" one) and s' (similar, but activating a buzzer instead of a LED), designed independently. If these two systems have to be deployed on the same micro-controller, how to compose them such as the button triggers both the LED and the buzzer ?
11	LCD screens	3	This kind of actuators receives messages (*i.e.*, strings) as inputs, and consumes more than a single pin when connected to a micro-controller. Enhance the expressiveness of the language to support such messages in addition to sensor values, and identify conflicting situations where the screen will interact with other pieces of hardware.

12	Macro definition	3	Enhance the DSL to support the definition of macros used to represent recurring behaviors.
13	Video game code generator	4	Define a new DSL able to model sequences of actions such as "UP, UP, DOWN, RIGHT" on a joystick. This DSL will be transformed into ArduinoML to support the recognition of such an action sequence by the micro-controller.
14	Models at runtime	4	Implement a communication layer through the serial port of the micro-controller to monitor the state of the model at runtime on a remote computer.
15	Sensor control dashboard	4	Leverage remote communication to collect the data sensed by the system and display it in real time in a web-based dashboard.
16	Verification & Validation	4	Use your favorite formal language to verify properties of the modeled systems (*e.g.* correctness, termination).

3 Illustrating Modeling Concepts with ArduinoML

The previous section described the ArduinoML project, and how it is implemented in our graduate program. This section focuses on the different dimensions of MDE that are illustrated through ArduinoML. We made the effort to assess that each of these dimensions is covered in the kernel, at least partially. Then, specific extensions allow students to emphasize a particular dimension and strengthen their kernel with respect to this dimension.

3.1 Domain Modeling

Domain modeling is the essence of this project. Students have to understand the way a micro-controller works (usually by following tutorials) and formalize the gap between what they want to express as users and what can be executed on a given micro-controller. One of the advantages of this project is the obviousness of the gap, especially when a professor who pretends to not understand anything too technical interacts with students. The result of this domain modeling phase is implemented in a meta-model acting as the backbone of their DSL.

Several extensions are related to domain modeling. For example, extension #1 introduces the notion of execution frequency in a given system, and extension #5 emphasizes the need to handle analogical sensors. This specific extension implies a shift in the way students design their systems, as they must be able to model conditions such as "*if the temperature is greater than 54 Celsius degrees, activate a safety alarm*" instead of simple boolean flags.

3.2 Domain Specific Languages

In ArduinoML, concepts behind DSLs are used to expose the work done at the domain modeling level as we follow a meta-model first approach. Students must deliver after only 2 weeks a preliminary domain model, which acts naturally as

the abstract syntax of their DSL. Then, they can spend the remaining 6 weeks to design a proper concrete syntax associated to their needs.

Three extensions are dedicated to DSLs. Extension #9 requires to define a querying language used to extract knowledge from the modeled systems. This extension allows students to think about how a model can be traversed and queried, and to properly define such a query language. Extensions #8 and #13 define minimalist but completely new languages, which need to be compiled to ArduinoML. These two extensions focus on the abstraction level related to DSLs, introducing a domain shift between ArduinoML and the new languages.

3.3 Constraints, Verification & Validation

The ArduinoML kernel needs to support elementary constraints, as for example two sensors cannot be connected to the same pin. However, some models can be correct with respect to the hardware but incorrect from a behavioral point of view (*e.g.* modeling situations one cannot escape from). Students must work on how these violations are handled by their system: is such an error fatal, or simply reported as a warning to the user?

Extensions #7 and #16 are related to constraints, verification and validation. The first one introduces "variable constraints", that is, constraints that are only activated under a given set of conditions. If it is always true that a pin cannot be shared by several hardware elements, the use of a buzzer instead of a LED is only needed when the system is dedicated to visually impaired people. Students must provide a way to express these variable constraints in their DSL, and to support it with the associated checker.

3.4 Model Transformation and Code Generation

The kernel includes a code generator for a minimal version of the language, supporting the illustration of model-to-text transformations. This transformation is described as a *necessary evil*, as the course does not focus on code generation. The course emphasizes the domain modeling part and the importance of focusing on the user's expressiveness. But as the modeled systems need to be executable, students understand quite easily the importance of investing into a code generator instead of implementing each system by hand. The low level of abstraction of the Arduino platform language (close to C, and without any level of abstraction dedicated to sensors and actuators at the syntax level) widens the created gap. It was clear while supervising the lab sessions that students understood the importance of modeling at the right level of abstraction, especially when the non-technical person was acting as a customer and was systematically rejecting technical details, focusing on the added value of using their DSLs.

The extension #10 focuses on model-to-model transformation, as it aims to define a composition operator able to merge two systems that need to be executed on the very same piece of hardware. The extension #6 is a blueprint generator, showing to the user how to connect its elements to the micro-controller. It emphasizes the fact that models can be exploited to generate things different than

programming code, such as documentation. Marginally, the extensions dedicated to the design of new DSLs on top of ArduinoML are related to model-to-model transformations principles, as they must generate valid ArduinoML models.

3.5 Variability

The ArduinoML kernel does not provide natively a support for variability modeling. To tame this issue, we rely on the extensions, as it is mandatory to deliver at the end a set of extensions (scoring at least 5 stars). Thus, we ask the students to deliver a report describing their kernel, and for each extensions what was the impact of introducing this extension into the kernel. It could then be possible to infer a software product line of their report, where the assets associated to each feature (an extension) are the associated section in their reports.

Two extensions focus more specifically on variability modeling issues. Extension #2 introduce the concept of hardware kits (built and sold by manufacturers such as Grove or SeeedStudio), acting as families of sensors one can use in a system. Students are asked to think about how these families can be represented, and how to guide the user while selecting the elements used in their system. For example, as the user already owns a Grove kit (this is the kit we bought to support the lab sessions), she will be guided to use the Grove equivalent of the sensor she selected from another manufacturer.

4 Conclusions & Perspectives

This experience report described how the ArduinoML project was implemented as part of our graduate curriculum in Computer Science. This case study leverages relatively cheap hardware kits (we invested around $700 to buy the hardware and equip the lab for up to 40 students) to allow the students to design and tool their own DSL, on top of third-party electronic equipment.

According to an ISO 9001 process, courses are self-evaluated by the students after the examination period. The survey sent to the student cannot be considered as scientific as the questions are imprecise and its completion by the student population is not mandatory. However, we think it is rather good to provide an overview of how the course was perceived by the students. 14 students out of 39 took time to respond, which is more than the usual response rate (36% instead of 10%). 86% of those questioned declared to be very satisfied by the project, and only one student was not happy with the call for bid principle, expecting a project description to be more guided. When asked to express what thay liked in the course, students answered statements such as "*interesting, useful and not boring*", "*very interesting project*", "*DSLs are fun!*", "*a very good way to bind theory to practice*" or "*the extension-based approach allows us to focus on what we found interesting in the domain*". However, some negative points remain, such as "*some extensions do not work well together and implies to work more than the other groups*", "*the workload is really important to implement the complete assignment, my group was badly organized and the final rush was tough*"

and "*as the project focuses on a single domain, if you do not like the case study, you spend the whole course working on something you do not really appreciate*". Based on this feedback, we plan for next year to assess the extensions chosen by the students as part of the initial feedback they receive after 2 weeks. To tame the workload, we will provide a set of milestones with respect to their meta-model and the chosen extensions, so they can plan work time according to these milestones. There is no real solution to the single-domain issue at the project level, so we decided to enhance the lectures to introduce additional case studies during the lectures.

Acknowledgments. The work reported in this paper is partly funded by the PING project[4] of the GDR GPL (CNRS, France). PING aims to collect and publish, at the national level, recipes on how to better teach software engineering.

References

1. Batory, D.S., Latimer, E., Azanza, M.: Teaching Model Driven Engineering from a Relational Database Perspective. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P.J. (eds.) MoDELS. Lecture Notes in Computer Science, vol. 8107, pp. 121–137. Springer (2013)
2. Blay-Fornarino, M.: Project-based Teaching for Model-Driven Engineering. In: MODELS Workshops. pp. 69–75. Educators Symposium, Warsaw University of Technology, Michal Smialek, Toulouse, France (Sep 2008)
3. Fleurey, F., Morin, B., Solberg, A.: A Model-driven Approach to Develop Adaptive Firmwares. In: Giese, H., Cheng, B.H.C. (eds.) 2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 168–177. ACM (2011)
4. Fowler, M.: Domain Specific Languages. Addison-Wesley, 1st edn. (2010)
5. Mosser, S., Fleurey, F., Morin, B., Chauvel, F., Solberg, A., Goutier, I.: SENSAPP as a Reference Platform to Support Cloud Experiments: From the Internet of Things to the Internet of Services. In: SYNASC. pp. 400–406. IEEE Computer Society (2012)
6. Paige, R.F., Rose, L.M.: Lies, Damned Lies and UML2Java. Journal of Object Technology 12(1) (2013)
7. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)
8. Whittle, J., Hutchinson, J., Rouncefield, M.: The State of Practice in Model-Driven Engineering. IEEE Software 31(3), 79–85 (2014)

Appendix: Technological Environment

At the hardware level, we used Arduino Uno micro-controllers[5] and predefined packs of sensors and actuators edited by Grove[6]. From a software point of view, we used classical tools from the modeling community, *i.e.*, the Eclipse Modeling Framework for the meta-modeling part, XText for the DSL implementation and the OCL for constraints implementation (Eclipse plugin). We plan to experiment with MPS[7] next Fall.

[4] `http://ping.i3s.unice.fr`
[5] `http://www.seeedstudio.com/depot/Arduino-Uno-Rev3-p-694.html`
[6] `http://www.seeedstudio.com/depot/Grove-Starter-Kit-p-709.html`
[7] `http://www.jetbrains.com/mps/`