Practicing Domain-Specific Languages: From Code to Models

Laure Gonnord, Sébastien Mosser

Practicing Domain-Specific Languages: From Code to Models

Laure Gonnord
Univ Lyon, Université Claude Bernard Lyon 1
LIP, CNRS, ENS de Lyon, Inria
F-69342, LYON Cedex 07, France
Laure.Gonnord@ens-lyon.fr

Sébastien Mosser
Université Côte d'Azur, CNRS, I3S, France
Sophia Antipolis
mosser@i3s.unice.fr

ABSTRACT

This paper describes our experience in constructing a new Domain-Specific Language course at the graduate level whose objectives is to reconciliate concepts coming from *Language Design* as well as *Modeling* domains. We illustrate the course using the reactive systems application domain, which prevents us to fall back in a toy example pitfall. This paper describes the nine stages used to guide students through a journey starting at low-level C code to end with the usage of a language design workbench. This course was given as a graduate course available at *Université Côte d'Azur* (8 weeks, engineering-oriented) and *École Normale Supérieure de Lyon* (13 weeks, research-oriented).

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Designing software**; *Development frameworks and environments*;

KEYWORDS

Software Modeling, Domain-Specific Languages, Code Generation, Code Abstraction, Reactive Systems

ACM Reference Format:
Laure Gonnord and Sébastien Mosser. 2018. Practicing Domain-Specific Languages: From Code to Models. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18 Companion), October 14–19, 2018, Copenhagen, Denmark*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3270112.3270116

1 INTRODUCTION

The foundation of the course we propose is the observation that the numerous concepts behind "Language Design" and "Software Modeling" are difficult to apprehend for students. From the point of view of educators, numerous difficulties arise when we build on a new instance of each of these two courses.

On the one hand, teaching language design (at both graduate or undergraduate level) is a necessary but painful task. Students struggle to understand why do they need to know how to design languages: writing a compiler (or an interpreter) is a tedious task, and the large number of existing programming languages ensure that at least one will implement the feature needed for a given purpose. In addition, this kind of courses are often implemented under the name "*Compilation*", and usually focus on lexical analysis, attributed grammars or symbol resolution [1, 8], and forgets the most relevant parts, namely, abstractions and semantics. It is then hard to focus on the part related to *Language Design* when students are ensnared in a difficult (from a theoretical point of view) and technical (from a practical point of view) context. However, we defend that in addition to the underlying foundations related to this field, it is important for student to understand how to *design* a language. It will help them to create their own when relevant, but also help them to classify existing ones and support their choices.

On the other hand, teaching modeling is also a necessary but painful task [5, 6]. We defend it as necessary considering that the essence of modeling is abstraction, *i.e.*, the ability to remove unnecessary details from a complex situation. But finding the right way to teach modeling is complicated [2]. Students might struggle with complicated technological stacks, syntactical issues in the UML [7] and have difficulties to understand the differences between models and meta-models when applied to simple toy examples. We defend that software developers must be confronted to modeling during their studies to identify the strength of abstraction-driven approaches. Clearly, a "modeling for modeling" approach does not work, and the infamous "UML to Java" example [4] cannot reasonably be used in 2018 to support model-based courses.

During a conference dedicated to software engineering and programming languages hosted in Besançon in 2016, the two authors met and exchanged the views described in the two previous paragraphs. During this discussion, it was clear that the main issue was to consider *Software Language Engineering* (SLE) and *Model-driven Engineering* (MDE) as two disjoint sets. We decided to leverage our experience in teaching languages and models to create a common syllabus for a course shared by *École Normale Supérieure de Lyon* (ENSL) and *Université Côte d'Azur* (UCA). This course uses the point of view of *Domain-specific Languages* (DSLs) to support the teaching of language design and abstraction, using a practical approach. We used as foundations a case study dedicated to embedded devices from a language point of view [3], coupled to our experience in teaching embedded systems from a reactive programming point of view. The goal of this paper is not to describe the syllabus of the course[1] but to share the rationale of the course, with the description of a lab session that goes through multiple level of abstractions and different technological stacks.

The paper is organized as follows: in Section 2 we depict the objectives we identified for this course, and the lab examples it is built on. In Section 3, we develop the objectives of each of the first steps of the "Minimal and Viable example", that illustrate different levels of abstractions, from code to models (of code). Then, in Section 4, we show alternative approaches working at the language

[1] Available online: https://github.com/mosser/sec-labs

MODELS '18 Companion, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-5965-8/18/10…$15.00
https://doi.org/10.1145/3270112.3270116

design level to illustrate modeling concepts. Finally, the last section (Section 5) gives more information about how the course was implemented in both universities.

2 COURSE OVERVIEW

The keystone of the pedagogical approach we follow is to use the predisposition of students to work with code to catch their curiosity and make them work on the concepts that drive this course: identifying abstractions to design languages. Considering DSLs as the object under study, the course we propose has then the following objectives:

- O_1 Illustrate how to abstract code into models;
- O_2 Identify how to operationalize models according to different targets (*e.g.*, ease of development, intended users);
- O_3 Study the relationships that exist between concepts and tools;
- O_4 Acquire experience in modeling through hands-on lab sessions.

The course is implemented in a "two-phases" fashion. The first phase relies on a *minimal and viable lab example*, developed in Section 3 and 4: the rationale of this phase is to discover and experiment the DSL main concepts in a guided way. The second phase consists in the creation of a complete language for a new domain in an unguided way. The approach we propose is entirely open source, with lightweight technology, and low cost embedded devices for the lab material. Thanks to this approach, the students progressively acquire the following knowledge and skills:

- The definition and practical use of the following concepts: model, meta-model (and their synonyms in language theory: languages, grammars), and object-orientation and reflexivity;
- A methodology to design a new language for a specific application domain: identify what is reusable, and what needs extensions, make rational implementation choices, test;
- An experience in designing a real-life DSL, targeting a business-driven case study.

Minimal & Viable Lab Example

The lab example we propose is based on the Arduino[2] open-source technology, and the use of some sensors and actuators: one 7-segment display, a button, a led per platform. The platform can be designed around a breadboard as in Fig. 1 or thanks to a pre-built Arduino shield (Fig. 2). An Arduino Uno micro-controller costs 20€. To build the breadboard version, one must buy small electronic hardware (a breadboard, a LED, a button and a display) for approximately 10€. The shield version is more expensive, as the shield can cost up to the price of the Arduino board according to vendors. A platform can be used by one group of up to four students, where two is the right team size based on our experience.

In this lab, we propose a sequence of "stages", each stage being built in the same "2-steps"way:

(1) Students are given a *minimal* working example (switching the LED on and off) of the language/technology used in the stage. They experiment and begin to criticize the solution in terms of performance, readability, usage, …

[2]https://en.wikipedia.org/wiki/Arduino

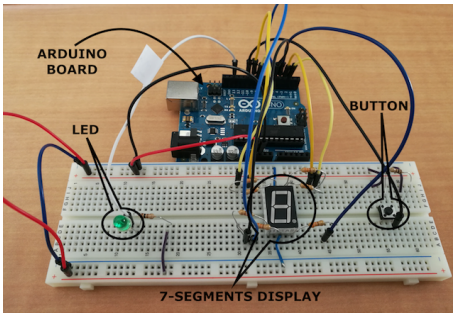

Figure 1: Arduino board and electronic breadboard

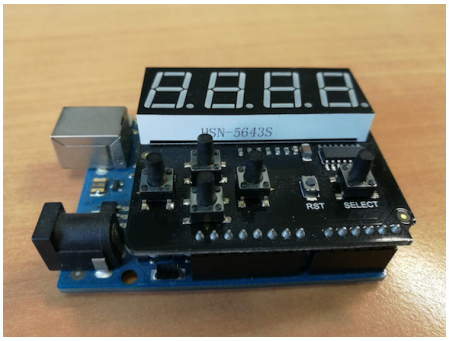

Figure 2: Using a pre-built shield on top of an Arduino board

(2) Then, they modify the example in a non trivial way to transform it in a *viable* example, representative of the domain. We here propose to use a 7-segments display to count time and reset. This example is non trivial since it requires to introduce memory states. The two applications need to be composed on the very same board: pushing the button changes the LED state, and also reset the counter to 0.

In the next two sections, we focus on the description of the nine stages identified in this minimal and viable example. The idea is to describe what we give to the students to kick-off the work for each stage, and the questions we use to drive the associated "step back" discussions and guide their report writing.

3 FROM CODE TO MODELS

To prevent falling back in the toy example pitfall, we illustrate the course using the *reactive systems* application domain. This domain is pertinent since there is a long tradition in designing *specialized* languages and development processes for these kind of software, especially in the area of *critical embedded systems*. Here we choose a less ambitious subdomain, namely, programming a micro-controller reacting to tiny sensors and operating on actuators. However, the modeling and code issues that arise from this simplified case are representative and a good abstraction of the real one.

This section illustrates both objectives O_1 and O_2 of the course. Indeed, the first three stages of the lab illustrate different levels of abstractions one can use while programming an Arduino-based reactive system. From these stages, we start discussions about tools and methods to gain abstraction in a piece of code (O_1) and also the pros and the cons of the different approaches according to

Listing 1: Minimal example: Plain C code

```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
  DDRB |= 0b00100000;
  while(1) {
    PORTB ^= 0b00100000; _delay_ms(1000);
  }
  return 0;
}
```

Listing 2: Minimal example: ArduinoLib code

```
#include <avr/io.h>
#include <util/delay.h>
#include <Arduino.h>

int led = 13;

int main(void) {
  pinMode(led, OUTPUT);
  while(1) {
    digitalWrite(led, HIGH); _delay_ms(1000);
    digitalWrite(led, LOW); _delay_ms(1000);
  }
  return 0;
}
```

many criteria (*e.g.*, expressivity, facilities to extend, maintain, debug, identity of the end-user), targeting O_2.

3.1 Plain Code (C)

The first stage uses C code working at the micro-controller registries level. We provide a running piece of code (Lst. 1), as well as the environment to compile it (thanks to `avr-gcc`) and upload the compiled image to the micro-controller (thanks to `avr-dude`) with a `Makefile`. The given code enables a LED plugged on pin 13 to blink forever at a frequency of 1Hz. At this low level of abstraction, accesses to sensors and actuators consist in injecting an electrical current in the micro-controller physical pins. At the code level, this is done thanks to parallel writes to ad-hoc registers called `PORTx` (x being B, C or D), that are configured with the help of the corresponding `DDRx` registers (input:0 or output:1). For instance, in Lst. 1 at line 5, we setup the LED plugged to pin 13 as an actuator by setting to 1 the 5^{th} bit of the `DDRB` registry (the first 8 pins being handled by `DDRA`, $13 = 5 + 8$). Then, thanks to an infinite loop, we switch it on and off with the help of a `xor` applied to the very same bit in the `PORTB` registry.

Considering this piece of code, students are asked to answer to the following questions and invited to elaborate and argument their answers on paper:

- What can we say about readability of this code? What are the skills required to write such a program?
- Regarding the application domain, could you characterize the expressivity? The configurability of the code to change pins or behavior? Its debugging capabilities?
- Regarding the performance of the output code, what kind of parallelism is expressed by the use of the `DDRx` registers?
- What if we add additional tasks in the micro-controller code, with the same frequency? With a different frequency?

3.2 Using the Arduino Library (C)

The second step uses the Arduino library[3] which is a C++ library provided by the Arduino designers. This library provides higher-level access to each pin individually, for example to support configuration in write or read mode with a function called `pinMode`, or to control the electrical current sent to a given pin with a call to `digitalWrite`. The code given to students is depicted in Lst. 2. This code is an iso-functional version of the one depicted in Listing 1, using the Arduino library. Based on the given example and the realization of the 7-segments counter, we ask students to discuss about the following questions in their report:

- Is the readability problem solved?
- What kind of parallelism can still be expressed?
- Who is the public targeted by this "language"?
- Is this language extensible enough to support new features? What is the price for the developer?

3.3 Programming a Finite State Machine (C)

The two applications to be developed for the micro-controller helps students to identify the need for abstraction when targeting a specialized domain. They easily identify that the LED and 7-segments functionalities can be modeled thanks to a *Finite State Machine* (FSM). Before moving to a model-driven approach, we use in this stage a convention-based approach to reify abstractions at the code level. Considering a system where one can express `transitions` between `states`, it is possible to implement such an FSM using functions as states, and conditional instructions coupled to terminal function call for transitions (see Lst. 3). Using this abstraction, we raise the following questions to help students understand the importance of abstraction at the code level:

- Does introducing a convention solve the readability issue?
- How to extend an app with a new feature? Does the approach prevent one to perform invasive changes in the existing behavior to introduce a new one?
- How to extend the code so that to support new features, *e.g.*, memory-less tasks, state-full tasks, different frequencies?

3.4 Modeling an ArduinoApp (UML & Java)

We use this stage to leverage the insights gained at the previous one, and emphasize the importance of working at the model level when dealing with abstractions. The idea here is to show firstly, that working with models free the user from the syntax, and secondly, that code generation mechanisms can be used to reach the previously defined operational target. Using a model-driven approach, it becomes clear to the students that the user is now restricted to the *vocabulary* available in the meta-model, and cannot deviate from it. It helps to position the meta-model as the abstract syntax of the language, defining *what* concepts needs to be exposed to the user. With respect to code generation, we show how the concepts

[3] https://www.arduino.cc/reference/en/

Listing 3: Minimal example: functional FSM

```
#include <avr/io.h>
#include <util/delay.h>
#include <Arduino.h>

int led = 13;

void state_on() {
  digitalWrite(led, HIGH); _delay_ms(1000);
  state_off();
}

void state_off() {
  digitalWrite(led, LOW); _delay_ms(1000);
  state_on();
}

int main(void) {
  pinMode(led, OUTPUT);
  state_on();
  return 0;
}
```

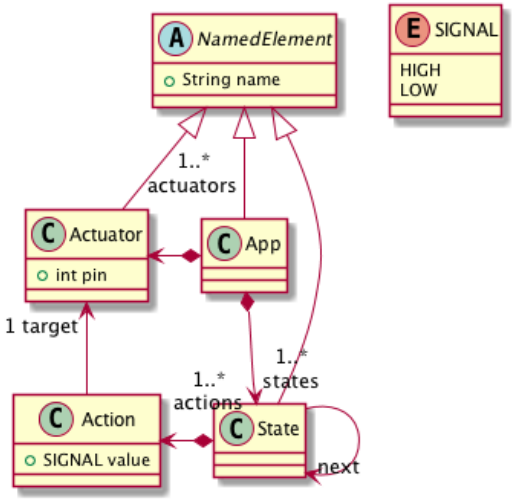

Figure 3: Minimal FSM meta-model

defined in the meta-model can be operationalized using classical object-oriented design patterns (*e.g.*, Visitor, Observer) to reach an executable target.

We provide to the students a minimal FSM meta-model, as depicted in Fig. 3. We also provide a running Java implementation of the meta-model and a simple Visitor implementation supporting naive code generation from a given model to C code (Lst. 4). We use the code manually written by the students during the previous stage to illustrate code templating and the underlying concepts associated to the Visitor design pattern.

Based on their extension of this example to support the 7-segments counter, we ask the student to discuss the following questions:

- What are the pros/cons associated to the meta-modeling approach? What is the cost of defining a meta-model? What is difficult in this activity?

Listing 4: Simple FSM Visitor implementation

```
public class ToC extends Visitor<StringBuffer> {

    @Override public void visit(App app) {
        c("#include <avr/io.h>");
        c("#include <util/delay.h>");
        c("#include <Arduino.h>");
        c("");
        c("void setup(){");
        for(Actuator a: app.getActuators()){
            a.accept(this);
        }
        c("}\n");

        for(State state: app.getStates()){
            h(String.format("void state_%s();",
                    state.getName()));
            state.accept(this);
        }

        if (app.getInitial() != null) {
            c("int main(void) {");
            c("  setup();");
            c(String.format("  state_%s();",
                    app.getInitial().getName()));
            c("  return 0;");
            c("}");
        }
    }

    @Override public void visit(Actuator actuator) { ... }
    @Override public void visit(State state) { ... }
    @Override public void visit(Action action) { ... }

    private void c(String s) {
        this.code.append(String.format("%s\n",s));
    }
}
```

- From the user point of view, what does it change? Is the approach usable for large apps?
- Consider the LED app and the counter one as two separate models. Is it possible to automate the creation of the final app based on these two models?
- What about the readability of the generated code compared to the previous one "by hand"? Its debugging capabilities ? Its extensiveness?
- Explain the interest of modeling in terms of genericity, functional property verification.

3.5 Remodeling an ArduinoApp (UML & Java)

During the previous stage, students understand quickly that working with the meta-model defined in Fig. 3 is not suitable for large applications: the final FSM is the cartesian product of the two apps (led and counter). We offer them here two choices:

- Creating a composition operator to support the combination of elementary applications to produce complex ones;
- Switch to another kind of abstraction that will provide a better support for end-users.

Listing 5: Composition operator skeleton in Java

```
public class CompositionLaw extends BinaryLaw<App>{
    @Override public App compose(App left, App right) {
        // ...
    }
}
```

Listing 6: Reactive version of the ArduinoLib code

```
#include <avr/io.h>
#include <util/delay.h>
#include <Arduino.h>

int led = 13;
bool is_high = false;

void led_change_state() {
  if (is_high) { digitalWrite(led, LOW); }
  else { digitalWrite(led, HIGH); }
  is_high = !is_high;
}

int main(void) {
  pinMode(led, OUTPUT);
  while(1) {
    led_change_state(); _delay_ms(1000);
  }
  return 0;
}
```

For the sake of concision, we will not focus in this paper on the contents of the composition method, as it is basically the implementation of classical FSM composition (considering shared actuators and sensors based on pin locations). We provide to the students a Java skeleton described in Lst. 5. We prefer to focus on the meta-modeling of the *reactive system* programming model, considering an infinite loop with a global state (memory). After reading the sensors, the new state is computed, then all actuators are updated. Listing 6 depicts a minor modification of Listing 2 following this paradigm. We provide to the students a starting meta-model for this paradigm, depicted in Fig. 4.

Students are then asked to write the extended state-full version, and discuss the following points in their report:

- Compare how this modeling solution and the previous one match the domain, especially regarding expressiveness and scalability.
- What is the cost (*e.g.*, modeling, code generation) of a new feature for the developer?
- What about scalability of the modeling paradigm itself?

3.6 Conclusions

In this section we depicted how, from low-level Arduino code, where a single programming model is promoted, we abstracted the domain-specific features by modeling it in different ways. We also showed that the choice of the modeling paradigm has a substantial impact on the expressivity and extensibility of further developments. Thanks to a code-first approach and a journey through abstraction levels, we manage to lead the students to a point where they recognize the value of models, and see the benefits of using such artifacts.

From now on, we will no longer change the abstraction level, but make a tour on different domain-specific language and meta-modeling paradigms that will permit to generate domain-specific code.

Listing 7: Minimal example: Lustre code

```
node cpt(reset: bool) returns (led_on: bool) ;
let
   led_on = false -> not(pre(led_on));
tel
```

4 FROM MODELS TO DSLS

Thanks to the previous stages, we are now working at the model level. The following stages explore how models relate with tools (O_3), and how such models and tools can target different users (O_2). Contrarily to the previous stages that were sequential, these stages are independent, as they address different paradigm in an hands-on fashion (O_4).

4.1 Integrating an existing DSL (Lustre)

In Section 3 we ended up with the conclusion that using a reactive system representation was a suitable way to model the domain in a scalable way, avoid costly code generation, without sacrificing expressivity or end-user usage. As a consequence, now that we came with this new paradigm for modeling, why not searching for an existing (possibly domain-specific) language implementing reactive systems, so that to reuse it for our particular purpose?

The Lustre[4] synchronous language was intended to be used for the design of *critical real-time embedded systems*. However, some educational-driven experiences have been made for real-time programming courses[5]. This stage takes inspiration from them.

The minimal code depicted in Lst. 7 illustrates the key feature of this DSL: only the functionality of the infinite loop is described, avoiding implementation details as well as non logical time. Here, the node describes the actuator "`led_on`" as a boolean output whose value is `false` during the first period, then the negation of its value during the preceding period (`pre(led_on)`) forever. The infinite loop depicted in Listing 8 (where `ctx` depicts the context, *i.e.*, the current state) is compiled from this description (and the desired frequency). The user should also encode the *glue code* in a separate file.

The compilation chain and its relationship with the application domain being depicted in Figure 5, the students are invited to argument their answers to the following questions:

- Who is the intended user for such a language?
- What is the cost of reusing this existing DSL for the developer in terms of code?
- What is the cost of adding a new task of our domain?
- Was is the cost of adding a new hardware target?

[4] http://www-verimag.imag.fr/The-Lustre-Programming-Language-and?lang=en
[5] http://perso.univ-lyon1.fr/thierry.excoffier/COURS/COURS/TEMPS_REEL/tr_lustre.pdf, http://laure.gonnord.org/pro/teaching/SysTR1516_M1/tr_lustre.pdf

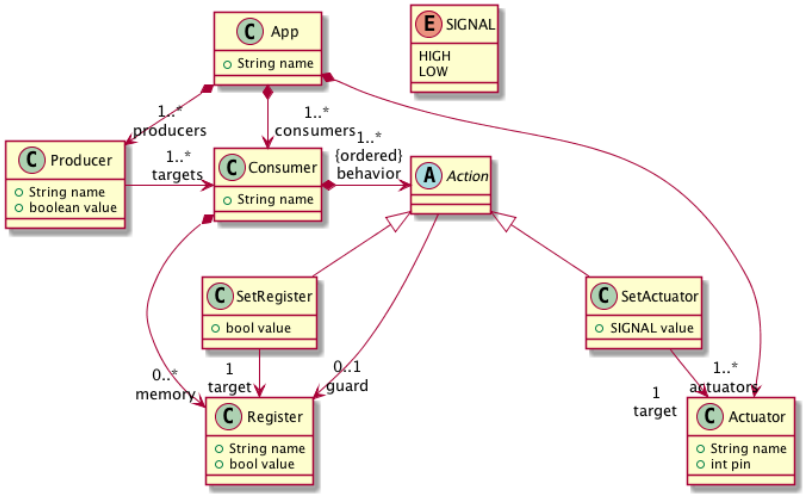

Figure 4: Minimal reactive meta-model

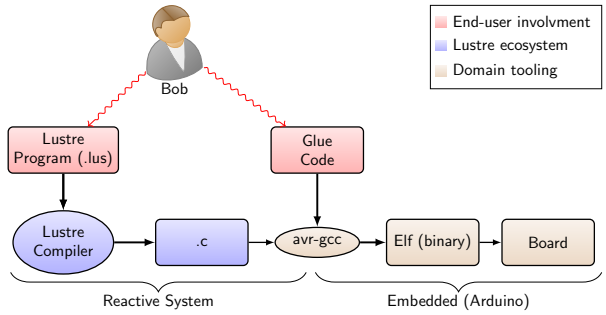

Figure 5: Lustre Compilation chain for Arduino

Listing 8: Generated C code from LUSTRE, and glue code

```
// cpt.c - Generated
void cpt_step(cpt_ctx* ctx){
  //...
  cpt_0_led_on(...);
}
// main.c - Generated
int main(){
  // ...
  while(1){
    cpt_step(ctx);
    _delay_ms(1000);
  }
   return 1;
}
//Glue code (Arduino target) - Hand written
void cpt_0_led_on(void* cdata, _boolean _V) {
  if (_V == 1)
    digitalWrite(led, HIGH);
  else
    digitalWrite(led, LOW);
}
```

- The LUSTRE language impose the memory to be bounded by construction. Is this a limitation for our (sub) domain?

Listing 9: Minimal example: Reactive code using ANTLR

```
App: Blinking

  led is an actuator bound to pin 13

  producer: quartz
    emit "tick" at 1Hz

  consumer: blinker
    bool state initialized as true
    state : led is HIGH
    !state: led is LOW
    state is !state

  blinker listens to quartz
```

- The LUSTRE language comes with its own ecosystem (test, formal verification), what are the generic properties we can imagine to prove from our domain?

4.2 Designing an External DSL (ANTLR)

The LUSTRE stage illustrates how to find and reuse an existing DSL that might fit a given purpose. In this stage, we ask the student to define a dedicated external language, reifying the domain concepts associated to their choice (FSM & composition or reactive system) directly in a dedicated syntax. We give to the student a kick-off implementation of an external grammar (using `Antlr` [6]), and an evaluation of the Abstract Syntax Tree that produce an instance of the previously defined meta-model (as a Java object). We also provide a program conform to the defined syntax (LST. 9), and the command line script to call the compiler and produce a reactive code associated to such a program.

Based on their extension of the grammar to support the counter application, students are asked to discuss the following points:

- Who is the intended user ? What about the tooling associated to the language?

[6] http://www.antlr.org/

Listing 10: Embedding the DSL inside the Scala syntax

```
object Switch extends ArduinoML {

  val button = declare aSensor() named "button" boundToPin 9
  val led = declare anActuator() named "led" boundToPin 12

  val on = state named "on" executing led --> high
  val off = state named "off" executing led --> low
  off.isInitial

  transitions {
    on -> off when (button is high)
    off -> on when (button is high)
  }
}
```

- More generally, what is the cost of such an approach?
- To what extent is the language fragile to the introduction of new features?
- What is the relationship between the meta-model and the grammar?
- How to validate that the defined syntax is the right one?

4.3 Designing an Embedded DSL (*e.g.*, Groovy)

Considering the cost of defining an external language, we explore here how to embed abstractions in an existing language instead of creating a new one from scratch. At this stage, we let the students free to chose their technological stack, an we only provide a link to the *embedded* directory of the ArduinoML zoo of syntax[7]. The idea of the ArduinoML zoo is to provide alternative syntaxes (11 embedded ones and 4 externals, provided by 9 contributors) to the FSM meta-model described in the previous section. Students can then pick-up one familiar language example in the zoo, and implement the counter application using their favorite language. When a language is not present, students are encouraged to publish a pull-request on the GitHub repository to update the zoo. Often, students chose the Groovy language to support their work at this stage, considering the large amount of documentation available (and maybe biased by their previous knowledge of Java).

Based on their work to adapt the ArduinoML example to the counter application, students are asked to discuss the following questions:

- How to chose between embedded or external?
- What is the impact of the host language choice?
- What about the maintainability of the concrete syntax?
- Who is targeted as an audience by this class of languages?

4.4 Using a Language Workbench (*e.g.*, MPS)

Considering the cost of designing an external language from scratch, and the intrinsic limitations of the embedded approach, we propose here to explore how dedicated workbenches can be used to model language. The key point here is to make students understand that language design is "simply" another domain, and that domain-specific tooling can be defined to support them, following the very same approach that they just use to support Arduino application designers. We chose the *Meta Programming System*[8] (MPS) to support this step, and also provide a link to the Xtext[9] version of the ArduinoML syntax for interested students. We give to the student a reference structure, and the associated projection to reach a concrete syntax (Fig. 6). Students can immediately use the generated environment and experiment code completion, syntax coloring, type constraints, which came for no additional costs.

When the implementation of the counter app is finished, we ask to the students the following set of questions:

- What is the cost/benefits ratio of using a workbench?
- What are the limitations of such an approach?
- What about vendor lock-in?

4.5 Conclusions

In this section, we described four versions of the same language, using alternative modeling approach to support its implementation. First, reusing a dedicated language helped us to discuss the concept of domain scope and model integration (through glue code). Then, we explored three different ways to create new languages capturing domain abstractions. These different ways help us to discuss, among others, domain evolution, meta-modeling principles, and user relevance.

5 LOGISTICS & EVALUATION

This course follows up a 5 years old course about Domain-specific languages taught at UCA. This new version is part of two different curricula: "Fundamental Computer Science" Master of Science at ENS and "Software Architecture" Master of Engineering at UCA. In Lyon, the format is 24 hours, supervised, including closely related lectures and labs (13 weeks, 4 credits). The course was attended by a small number of students which never attended any software engineering course, and are inexperienced in language design. However, they have a broad knowledge in semantics and program abstractions. At UCA, the course is classically attended by a large number of students (35), and lengths 8 weeks for 2 credits. The evaluation differs, as UCA values an engineering approach (thus evaluating a project) and ENSL is a research-oriented environment (half of the evaluation is made on a bibliographic study about DSLs, models and languages). We consider as a prerequisite basic notion of software development and modeling.

5.1 Case study examples for Phase #2

The nine different implementations of the reactive language are used in the course to support the first phase, where students explore in a guided way how to work with abstractions on a given domain. As educators, we guided their journey by providing reference code, and step back questions. The second phase of the course relies on the capture of a new domain, based on the experience learned during the first phase. We briefly describe here several cases studies used in the past to support DSLs and meta-modeling teaching.

- Sensor simulation: create a language supporting the modeling of sensors to support load testing of data collection middleware. Students have to model sensors based on polynomial interpolations, Markov chains, replay from legacy

[7] https://github.com/mosser/ArduinoML-kernel/tree/master/embeddeds

[8] https://www.jetbrains.com/mps/

[9] https://www.eclipse.org/Xtext/version

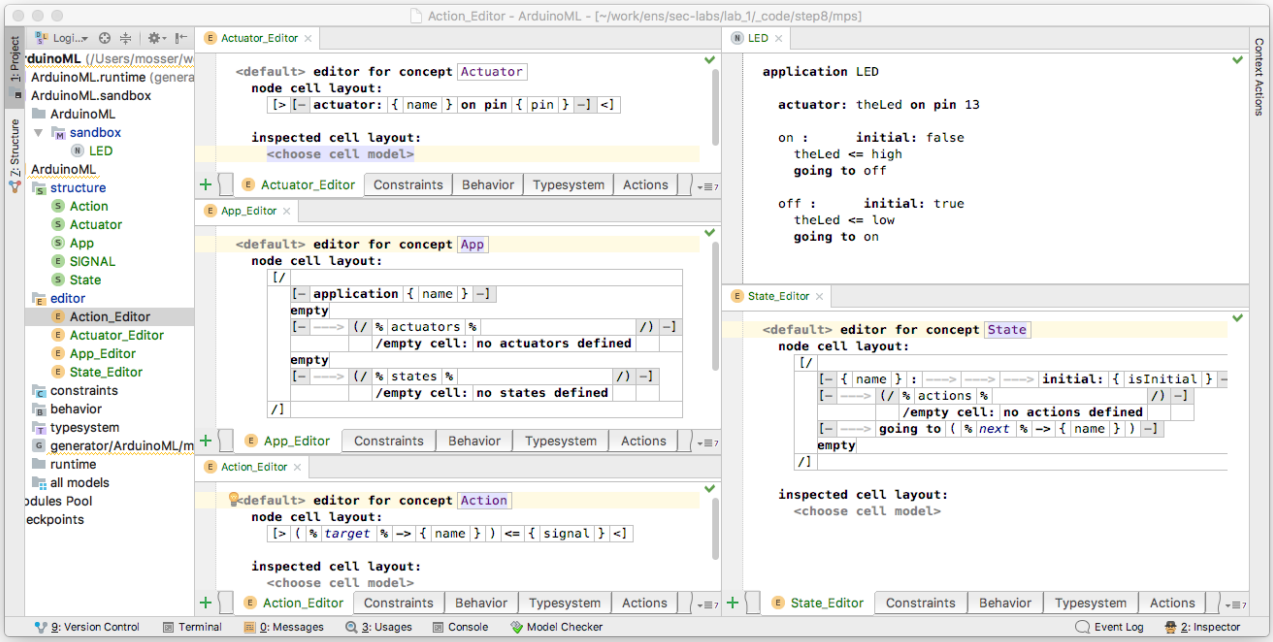

Figure 6: Modeling a concrete syntax using MPS

dataset, and define an execution environment to send the simulated data to a time-series database (*e.g.*, InfluxDb).

- Application deployment: create a language to support the deployment of services in a distributed environment. Students have to capture what is a service, how services relate to each others, create a deployment plan and upload the different codes to the modeled topology in order to setup a running ecosystem.
- Scientific Workflow: create a language to support the modeling of scientific workflows (*e.g.*, grid computing data processing, machine learning workflow). Students have to capture concepts like data sources, sinks, processors, and data links to transfer data among processors. They must also reach an execution context that respect the expected semantics for data flows.

6 PERSPECTIVES

The course is re-offered at UCA and ENSL for the next academic year. Discussions have started to implement it at *Université du Québec à Montréal* in the upcoming years at the graduate level. We plan to clean up the available material published on GitHub, which is for now scattered among several repositories (one instance per course and the ArduinoML zoo) into a single one. We also plan to start communicating about this course in the model-driven engineering and software-engineering communities to gather feedback from researchers and improve the lab contents. An in-depth evaluation of the course outcome is an ongoing work, as we plan to better evaluate this point in the new instances of the course.

Acknowledgments

Authors want to thanks the GdR GPL to support collaboration between researchers in France at the national level, which allow the creation of such inter-universities initiative. We also want to thanks Benoît Combemale for the fruitful discussions we had that helped to classify the different stages of the labs.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Don S. Batory and Maider Azanza. 2017. Teaching Model-driven Engineering from a Relational Database Perspective. *Software and System Modeling* 16, 2 (2017), 443–467. https://doi.org/10.1007/s10270-015-0488-7

[3] Sébastien Mosser, Philippe Collet, and Mireille Blay-Fornarino. 2014. Exploiting the Internet of Things to Teach Domain-Specific Languages and Modeling: The ArduinoML project. In *Proceedings of the MODELS Educators Symposium co-located with the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 29, 2014. (CEUR Workshop Proceedings)*, Birgit Demuth and Dave R. Stikkolorum (Eds.), Vol. 1346. CEUR-WS.org, 45–54. http://ceur-ws.org/Vol-1346/edusymp2014_paper_3.pdf

[4] Richard Paige and Louis Rose. 2013. Lies, Damned Lies and UML2Java. *Journal of Object Technology* 12, 1 (Jan. 2013). (editorial).

[5] Volkhard Pfeiffer. 2016. Teaching Domain-Specific Language Engineering and Model-Driven Software Development: A Competence-oriented Approach. In *Proceedings of the 12th Educators Symposium (EduSymp 2016).* 19–26.

[6] Alfonso Pierantonio, Federico Ciccozzi, Michalis Famelis, Gerti Kappel, Leen Lembers, Sebastien Mosser, Richard Paige, Arend Rensink, Rick Salay, Gabriele Taentzer, Antonio Vallecillo, and Manuel Wimmer. 2018. How do we teach Modelling and Model-Driven Engineering? A survey. In *Proceedings of the MODELS Educators Symposium 2018.*

[7] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. 2015. *UML@Classroom: An Introduction to Object-Oriented Modeling.* Springer Publishing Company, Incorporated.

[8] Linda Torczon and Keith Cooper. 2011. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.