ÉCOLE POLYTECH'NICE SOPHIA
SI5
2019-2020

Domain Specific Language

ArduinoML

Project delivery

Two different projects have to be delivered.
The first one should provide the basis arduinoML and (at least) one extension in both an internal and an external DSL. From the delivered DSL, it should be possible to write programs from which an arduino compliant code should be generated.
The second project will include the implementation in either an internal or an external DSL of another language that will be announced after the first delivery.
Deliveries are expected by email (to Julien Deantoni: firstname.name@univ-cotedazur.fr, with [DSL] as object prefix). First delivery is expected before **before** 02.02.19, 10:00PM Paris Time and the second delivery before 27.02.19 10:00PM Paris Time. The delivery is expected as a pdf report (please, do not write a novel, but use few words in a clever, effective and scientific way!). The report must contain :

- the name of people in your teams
- a link to the code of your DSL(s)
- a description of the language(s) developed:
  - The domain model represented as a class diagram;
  - The concrete syntax represented in a BNF like form;
  - A description of your extension and how it was implemented;
- a set of relevant scenarios implemented by using your language(s) (internal and/or external);
- A critical analysis of (i) DSL implementation with respect to the ArduinoML use case and (ii) the technology you chose to achieve it;
- Responsibility of each member in the group with respect to the delivered project.

Objectives: Define the ArduinoML language

Your objective here is to enhance the kernel available in the ArduinoML zoo[1] with new features, and deliver a fully-fledged DSL that add value for the final users of ArduinoML. You are asked to develop two implementations of the very same language: one using an external approach (e.g., AntLR, Lex/Yacc, MPS, XText, Sirius, Racket) and the second one using an embedded one (Groovy, Ruby, Java, Scala, ...). On top of a common kernel, you are asked to introduce in your language one extension (among seven) described at the end of this document.

[1] `https://github.com/mosser/ArduinoML-kernel`

Basic scenarios

1. **Very simple alarm**: Pushing a button activates a LED and a buzzer. Releasing the button switches the actuators off.
2. **Dual-check alarm**: Pushing a button will trigger a buzzer if and only if two buttons are pushed at the very same time. Releasing at least one of the button stop the sound.
3. **State-based alarm**: Pushing the button once switch the system in a mode where the LED is switched on. Pushing it again switches it off.
4. **Multi-state alarm**: Pushing the button starts the buzz noise. Pushing it again stop the buzzer and switch the LED on. Pushing it again switch the LED off, and makes the system ready to make noise again after one push, and so on.

Commmon Parts

The following parts must be available in your DSL:

- **Abstract syntax**: The abstract syntax should be clearly identified in the delivered code.
- **Concrete syntax**: The concrete syntax (external or embedded) must be clearly identified and used by a relevant set of scenarios. The syntax must leverage the tool chosen to implement it to make it clear and easy to use.
- **Validation**: Support your end-user by checking that a model is realizable on the Arduino platform. For example, compatibility of digital bricks with the pin used, termination of the modelled behavior, ...
- **Code generation**: Provide a generator producing turn-key arduino code, implementing the behavior. This code can be copy-pasted into the Arduino IDE and uploaded to a micro-controller.

"À la carte" features

The remainder of this document describes seven extensions that you can implement on top of your kernel. Each extension is defined by a short description and at least one acceptance scenario, and a difficulty defined by a number of stars. Chose (at least) one feature to introduce in your project.

Specifying Execution Frequency ($\star$)

The ArduinoML language works in continuous mode, i.e., the microcontroller continuously executes the deployed sketch. If the board is connected to a battery, it will eventually lead to empty it in a very short time. Considering situations where the board is used to sense data in a 24/7 way (e.g., a weather station capturing temperatures and light levels), one wants to specify the execution frequency.

> Acceptance Scenario: one wants to express that the modelled sketch has a given frequency (e.g., 0.5Hz). At the technical level, this frequency is transformed into "waits" between each transition (e.g., if f = 0.5Hz, the system stays 2 seconds in each state). This "wait" must be compatible with the ones defined by the user in each state: if one asks to wait for 200ms in each state, the system will only wait 1800ms to achieve a 0.5Hz frequency (writing and reading pin values are considered negligible). A warning can be detected if the wait actions introduced by the user conflict with the modelled frequency.

Remote Communication ($\star$)

The Arduino Uno board supports serial communication with a host computer. As a user, one can use ArduinoML (i) to send data from the sketch to the computer and (ii) to receive data from the computer and use it in the sketch. The Serial Monitor available in the Arduino IDE is a good tool to support such an interaction.

> Acceptance Scenario: By activating remote communication in her model, Alice uses the serial port as a sensor and interact with her sketch by entering data on her keyboard.

Exception Throwing ($\star$)

To implement this extension, we assume that a red LED is always connected on a given port (e.g., D12). One can use ArduinoML to model erroneous situations (e.g., inconsistent data received, functional error) as special states. These error states are sinks, associated to a given numerical error code. When the sketch falls in such a state the red LED blinks conformingly to the associated error code to signal the error to the outside world. For example, in an "error 3" state, the LDE will blink 3 times, then a void period, then 3 times again, etc.

> Acceptance Scenario: Consider a sketch with two buttons that must be used exclusively, for exemple in a double-door entrance system. If the two buttons are activated at the very same time, the red LED starts to blink and the sketch is blocked in an error state, as the double-door was violated.

PIN allocation generator ($\star$)

The ArduinoML kernel assumes that the user knows how a given bricks can be connected to the board. This assumption is quite strong, as the chassis' datasheet (see on my website) implies several technical constraints. For example, analogical bricks can be used as input on pins A1-A5, and pins D9-D11 supports analogical outputs as well as digital inputs and outputs. Using the BUS pins also impacts the available other digital or analogical pins.

> Acceptance Scenario: As a user, one can use ArduinoML to only declare the needed bricks in the sketch, without specifying the associated pins. The ArduinoML environment will perform the pin allocation with respect to the technical constraints, and describe it in the generated code (e.g., file header comment).

Supporting the LCD screen ($\star$)

The Electronic Bricks kit comes with an LCD screen. As an ArduinoML user, one can use the language to write text messages on the screen. These messages can be constants (e.g., "Hello, World!"), or built based on sensors or actuator status (e.g., "push button := ON", "temperature := 25 deg", "red light := ON"). One also expects the language to statically identify messages that cannot be displayed (e.g., too long). Moreover, using the bus connection prevent the use of several pins on the board (see the bus datasheet, on my website).

> Acceptance Scenario: The value sensed on the temperature sensor is displayed on the LCD screen.

Handling Analogical Bricks ($\star\star$)

The kernel only supports digital bricks. As a user, one can use the ArduinoML language to use analogical bricks. For example, the temperature sensor delivers the room temperature. Thus, analogical values can be exploited in transitions (e.g., if the room temperature is above 35 Celsius degrees, trigger an alarm). Analogical values can be set to analogical actuators (e.g., the buzzer or a LED), as constants or as values transferred from an analogical sensor.

Acceptance Scenario: considering a temperature sensor, an alarm is triggered if the sensed temperature is greater than 57 celsius degrees (fire detection).

Macros definitions ($\star\star\star$)

The ArduinoML kernel is purely declarative for now. It does not contain any abstraction to factorise "recurren" situations. In the given scenarios, the two states involve the same actuators (i.e., the red led and the buzzer). The first state deactivates the two actuators, and the second one activates them. As an ArduinoML user, one can use the language to model such "situation" (e.g., sending a given signal to the led and the buzzer). For example, the previously described situation involves the two actuators, and a "paramete". One can activate this macro to activate (or deactivate) both actuators at the same time.

Acceptance Scenario: Using this feature, one can model a blinking behaviour using a macro that activate and deactivate a set of actuators according to a given frequency.