POLYTECH NICE-SOPHIA

Université Nice Sophia Antipolis

Membre de UNIVERSITÉ CÔTE D'AZUR

Arduino-ML
Domain Specific Languages

Team ADSL2

BARNA Anthony
BURETTE Leo
DEFENDINI Lara
SAVORNIN Guillaume
VAN DER TUIJN Anton

ACADEMIC YEAR 2021 / 2022

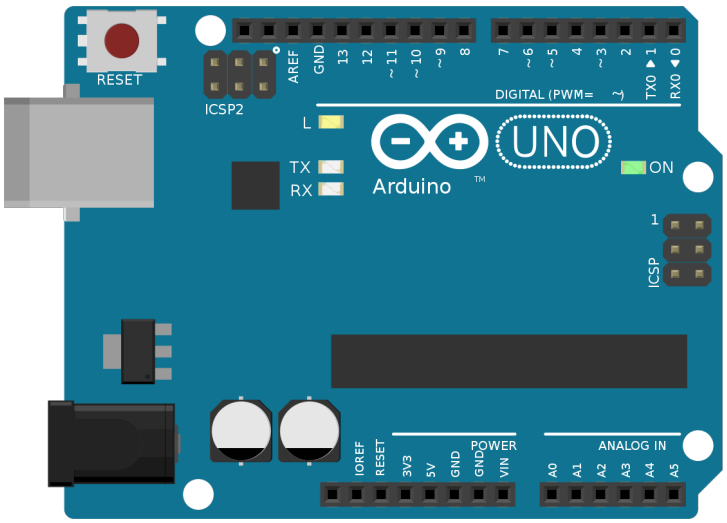


Table of Contents

The use of IoT devices becomes a very popular trend these days. The massive implementation of these devices has created new challenges such as allowing non-programmers to easily create Arduino programs. Inspired by this problem, the goal of our project is to be able to meet this need using the DSL approach.

This document presents the work produced by our group for the Domain Specific Language, DSL course. For this project we decided to implement a solution using the external DSL MPS and internal DSL Groovy.

The corresponding source code was hosted on GitHub at this address: https://github.com/Anton-vanderTuijn/dsl-ArduinoML

1 Domain Model

We updated our semantic model before each development to ensure that on one hand our choices were consistent and, on the other hand, that we could share a reference document on the upcoming changes with the rest of the team.

On the following diagram we represented the different types of elements by using colors:

- yellow for the Interfaces
- blue for the Classes
- green for the Enums

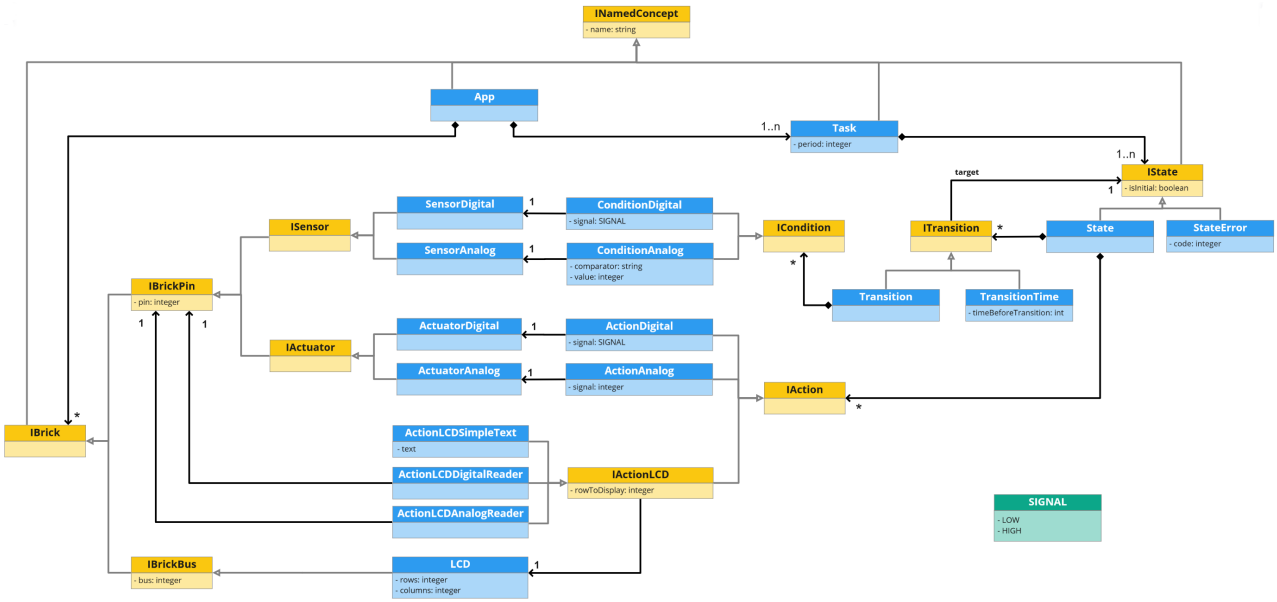


Figure 1: *Domain Model Diagram*

The App is the root of the Domain Model, it contains all the bricks defined by the expert. It extends the INamedConcept interface like some other models so that the user can understand what each of them means, and he can better reference them in the grammar.

The program contains multiple Final State Machines (FSM) with some states having transitions to other states.

The rest of the Domain Model will be explained in each extension's description.

2 Syntax: Extended Backus–Naur Form

In the following two sections, you can find an Extended Backus–Naur Form diagram for our external and internal DSL. We have a BNF for each DSL because the internal DSL is limiting too much the external one. Making only one would require having to strip off the syntax of the external DSL.

However, we tried to approach the external DSL's syntax with, as most as possible, internal DSL's mechanisms. Please refer to the section Retrospective to know more about the limitations of the languages we used.

The conventions used to write the BNF can be found in the Appendix A.

PS: The following words refer to primitive types: Boolean, String and Integer.

2.1 External DSL

```
Brick =
    (("Sensor Digital" | "Sensor Analog" | "Actuator Digital" | "Actuator Analog") String "on pin" Integer)
    | ("LCD" String "(" Integer "cols," Integer "rows ) on bus" Integer);

Task =
    "Task" String ":"
        "Period :" Integer "ms"
        (State)+
        (StateError)*;

State =
    "State" String ":"
        "initial :" Boolean
        "actions :" (Action)*
        "transitions :" (Transition)*;

StateError =
    "Error State" String ": error code:" Integer;

Action =
    (String "becomes" (Integer | Signal))
    | ("print" ((String "value") |  ("'" String "'")) "on" String "row n°" Integer);

Transition =
      "to" String ("when" (Condition ("and" Condition)*)) | ("after" Integer "ms");

Condition =
    String ("becomes" Signal) | (("=="|"!="|">="|"<="|">"|"<") Integer);

Signal =
    "high" | "low";

App =
    "Application" String
        (Brick)*
        (Task)+;
```

Figure 2: *MPS external DSL - BNF*

2.2 Internal DSL

```
Brick =
    (("sensorDigital" | "sensorAnalog" | "actuatorDigital" | "actuatorAnalog") String "pin" Integer)
    | ("lcd" String "cols" Integer "rows" Integer "onBus" Integer);

Task =
    "task" "{"
        "taskName" String
        "period" Integer
        (State)+
        (StateError)*
    "}";

State =
    "state" "{"
        "name" String
        "initial" String
        "actions" "{"
            (Action)*
        "}";
        "transitions" "{"
            (ToState)+
        "}";
    "}";

StateError =
    "stateError" "{"
        "name" String
        "code" Integer
    "}";

Action =
    ("actionDigital" String "becomes" Signal | "actionAnalog" String "becomes" Integer)
    | (("printText" | "printDigital" | "printAnalog") String "valueOn" String "row" Integer);

ToState =
    "toState" String "when" Condition ("and" Condition)*;

Condition =
    String (("becomes" Signal) | (("==" | "!=" | ">=" | "<=" | ">" | "<") Integer));

Signal =
    "high" | "low";

App =
    (Brick)*
    (Task)+
    "application" String;
```

Figure 3: *Groovy internal DSL – BNF*

2.3 Syntax Examples

```
application  Dual-check alarm

  Actuator Digital buzzer on pin 8
  Sensor Digital btn1 on pin 9
  Sensor Digital btn2 on pin 10

  Task program :
    Period : 1 ms

    State alarm_off :
      initial : true
      actions :
        buzzer becomes low
      transitions :
        to alarm_on when btn1 becomes high and
                         btn2 becomes high

    State alarm_on :
      actions :
        buzzer becomes high
      transitions :
        to alarm_off when btn1 becomes low
        to alarm_off when btn2 becomes low
```

```
sensorDigital "button_1" pin 9
sensorDigital "button_2" pin 10
actuatorDigital "led" pin 12

task {
    taskName "task"
    period 1000

    state {
        name "off_1"
        initial "true"
        actions {
            actionDigital "led" becomes "low"
        }
        transitions {
            toState "on" when "button_1" becomes "high" and "button_2" becomes "high"
        }
    }

    state {
        name "off_2"
        initial "false"
        actions {
            actionDigital "led" becomes "low"
        }
        transitions {
            toState "on" when "button_1" becomes "high" and "button_2" becomes "high"
        }
    }

    state {
        name "on"
        initial "false"
        actions {
            actionDigital "led" becomes "high"
        }
        transitions {
            toState "off_1" when "button_1" becomes "low"
            toState "off_2" when "button_2" becomes "low"
        }
    }
}

application "Dual check Alarm"
```

Figure 4: *MPS and Groovy syntax examples*

3 Implemented Extensions

3.1 Exception Throwing

We decided to add an abstraction layer to handle multiple types of states. Using an interface, we support the existing states (State) and the new exceptions states (StateError). The StateError has a variable named "code" containing the error code given by the user when defining the state. Moreover, this state is a final state meaning it does not have a Transition, meaning it will remain in the same state until the user decides to reset the Arduino. When the program throws an exception, it will display the error code by blinking the debug LED.

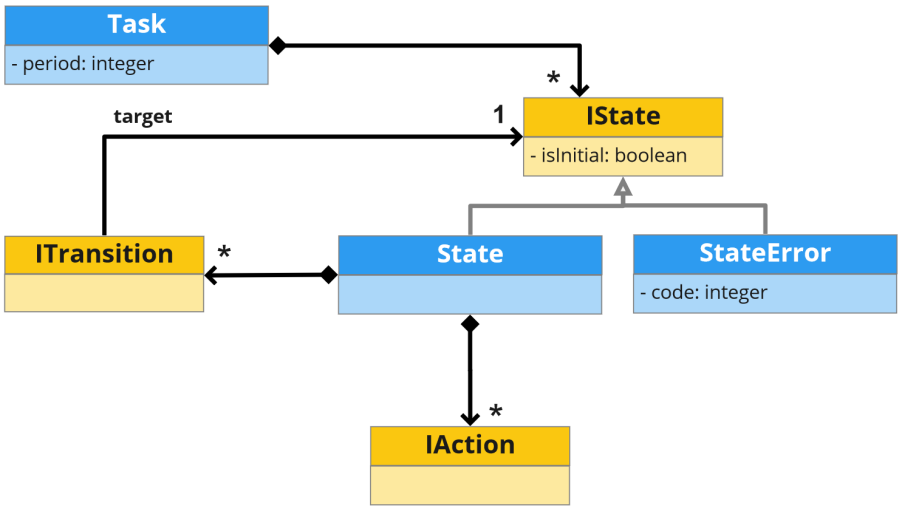


Figure 5: *Exception Throwing implementation*

3.2 Temporal Transitions

To support the Temporal transitions, we abstracted the Transition with an interface. Making it so, we now have two types of transition: Normal one and temporal one. The State is now referring to the interface, allowing it to have multiple type of transitions.

A Time Transition has access to the target State as a normal Transition, but instead of having multiple condition, it only needs an attribute indicating the time before transitioning to the other State.

Code generation wise, the temporal transition cannot block the other transitions, it still requires other types of transition to be accessed if they are valid.

This extension could also have been seen from the perspective of a Temporal condition. Making it possible to have a condition on a button possible only after a set of time. It would have allowed for more freedom for the expert, but it would have made the domain less intuitive.

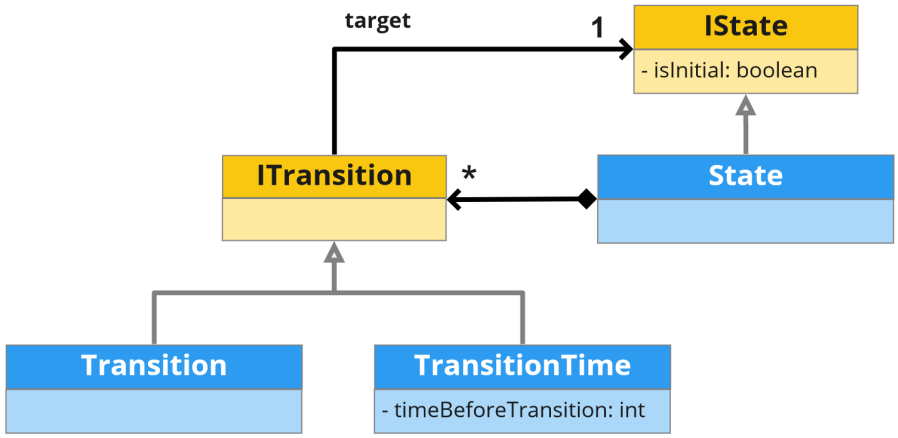


Figure 6: *Temporal transitions implementation*

3.3 Handling Analogical Bricks

Originally, no Analogical bricks were supported. To implement this extension we had to create an abstraction layer for the sensors (ISensor) and the actuators (IActuator). We also have an interface on top of ISensor and IActuator called IBrickPin (explained in section Supporting the LCD Screen).

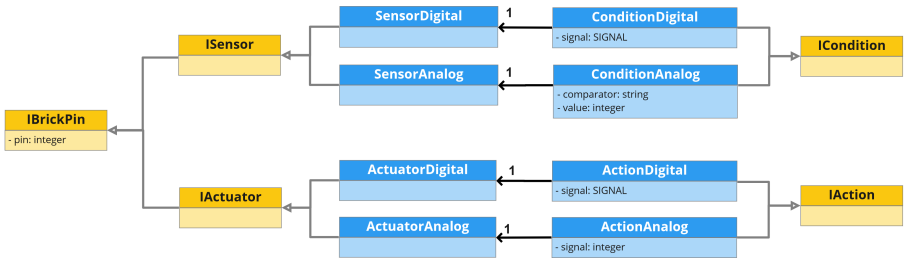


Figure 7: *Digital and Analogical implementation*

The ISensor interface is implemented by the SensorDigital and SensorAnalog class. In the same way, the ActuatorDigital and ActuatorAnalog implement the IActuator interface. We had to abstract the Conditions (ICondition) and the Actions (IAction) to handle the digital and analogical specificities.

To go further in the extension, we did not implement it, but it should be easy to give the user the possibility to calibrate the analogical bricks thanks to our domain model.

3.4 Supporting the LCD Screen

The LCD Screen works by being plugged to a bus, contrary to the other sensors and actuators. In the domain, it made us add an abstraction layer of different types of Bricks: the ones that are Pin based implementing the IBrickPin interface, and the ones that are Bus based implementing the IBrickBus interface. Both are themselves implementing the IBrick interface to allow bricks declaration in the App.

Now, the LCD Screen can just implement the IBrickBus interface. It also contains rows and columns attributes to support the different screen sizes.

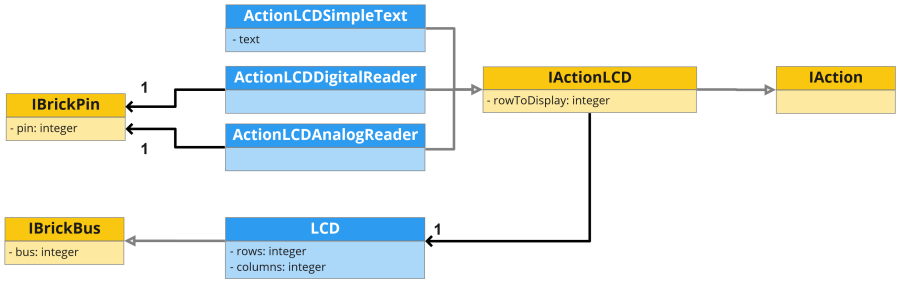


Figure 8: *LCD screen implementation*

Action wise, the expert needs to print values on the screen. That made us implement the IAction interface with another interface IActionLCD dedicated to the LCD.

We designed our model so that the expert can select on which row of the screen to display the value, making it easy to use and to display multiple things at the same time. This attribute can be found in the IActionLCD interface.

We made three different types of actions for LCD:

- **ActionLCDSimpleText:** print a text.
- **ActionLCDDigitalReader:** read the value of a pin-based brick in a digital way and print it alongside the brick's name.
- **ActionLCDAnalogReader:** read the value of a pin-based brick in an analog way and print it alongside the brick's name.

A limit to this implementation is that the expert can't create multiple LCD actions on the same row because it will overwrite it.

Reading an analog value for a digital sensor/actuator is possible in our domain model, in the same way as the other way around. This could have been constrained to the same signal type if an Analog or Digital Interface would have been created.

3.5 Parallel Periodic Region

Having multiple State Machine running at the same time requires to make changes directly below the root of the Application: instead of the App creating the States we chose to have a new model named Task to create them. A Task represent a single Final State Machine.

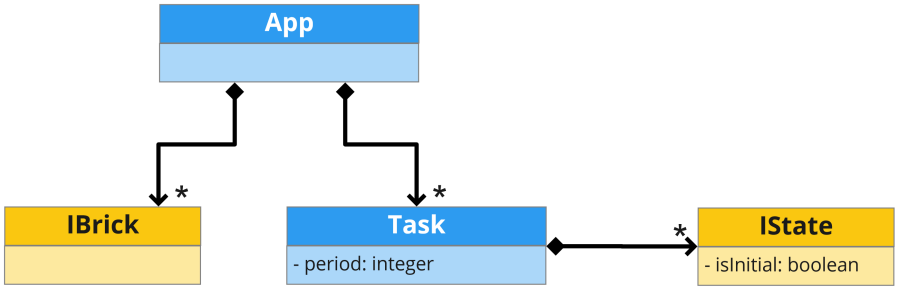


Figure 9: *Parallel Periodic Region implementation*

Because we work on an Arduino Uno, parallelism is not possible, we only do concurrency. That means the TaskScheduler library we use requires a period at which each task is called again. If we want to have a real time program the best, we can do is set the period for each task at 1 ms, but if a task takes longer than that the others will be delayed.

This also means the Error Handling extension need some changes, we originally did it using delays in the Arduino code, but here each delay would impact all the other tasks. By changing the waiting time from sleeping to busy waiting the other tasks can now run without interruption. (This update has only been made in MPS)

The limit is that if multiple error states are active at the same time, the expert will have a hard time reading the error because the error led is not customizable. This can be an improvement for the future.

4 Scenarios

4.1 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:** It will trigger a buzzer if and only if two buttons are pushed at the very same time. Releasing at least one of the buttons 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.

4.2 Extension Acceptance Scenarios

5. **Exception Throwing:** Consider a sketch with two buttons that must be used exclusively, for example 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.
6. **Temporal transitions:** Alan wants to define a state machine where led1 is switched on after a push on button btn1 and switched off 800ms after, waiting again for a new push on btn1.

7. **Supporting the LCD screen:** The digital value of a button actuator is displayed on the LCD screen as specified in the model.

8. **Handling Analogical Bricks:** Considering a light sensor, an alarm is triggered if the sensed light sensor's value is below 30.

9. **Parallel periodic Region:** Bob wants to check the state of a button btn1 every 400ms to toggle led1, and he also wants to check the state of btn2 every 50ms to switch on led2 when pushed and switch it off when released.

4.3 Extension Additionnal Scenarios (MPS only)

10. **Parallel Program with Error and Analog:** remake of the Analog and Exception acceptance scenarios: on one program if a light sensor's value goes below 30, then the error state (code 3) is active. While on another program a button "btn" can turn on or off an LED.

11. **Parallel Program with Time Transition:** On one program, a LED "led1" is turning on and off every 1000ms. While on a second program a button "btn" can be used to turn on and off a second LED "led2".

12. **LCD Screen Demo Alarm:** An LCD screen displays the value of a light analog sensor and of a digital led. If the light sensor's value goes below 30 the led turns on, if it goes back up to 40 it turns on. The two bricks values can be seen in real time on the screen.

5 Retrospective

5.1 DSL Implementation

Our team started by implementing the DSL using MPS and Groovy in parallel leading to minor differences in the Domain Model, and some extra features were supported by one implementation a not the other. As a remedy, we decided which feature will remain in both implementations and ensure that the Domain Model is the same. To implement the scenarios, we used and drawn state machines, it made it easier to understand an ensure the scenarios were implemented the same way in both languages. We started by implementing the base scenarios first, and after that we added the extension scenarios one by one. At the end, all the base and extension scenario were implemented in both DSLs.

Concerning the Groovy implementation, we kept the initial kernel base and modify it accordingly to our Domain Model. Unlike MPS we started our implementation from scratch using Mosser tutorial.

5.2 Technologies

5.2.1 JetBrains MPS

We chose to establish our external language using the MPS tool since it provides a lot of guidance and makes it simple to identify external languages. Furthermore, because the scenarios must adhere to a tight framework, their writing is substantially simplified.

The biggest disadvantage is that a language created with MPS is extremely reliant on the IDE. Because it cannot be opened by any other IDE, the language becomes far less practical without it.

But constraints are easy to implement, warnings and errors checking are displayed in real time as seen in the Appendix MPS Constraints: Errors and Warnings. Other things that make MPS a good alternative compared to the other external tools is the syntax coloring that can be heavily customized.

5.2.2 Groovy

We chose Groovy as our internal language because it is simple to integrate with Java and allows us to create a simple and flowing syntax with minimal understanding (no need to declare a class or a main function as language user). On the other hand, obtaining a rules file that can be used as an extension in an IDE or text editor to provide autocompletion and a linter is difficult. Furthermore, interpreting the files in accordance with the language necessitates the use of a JVM-based compiler. This necessitates the installation of java in addition to the compiler for our users.

We also faced multiple syntax limitations with Groovy. For instance, it does not support multi lines instructions easily, we had to use brackets to create scopes and be able to write on multiple lines. It is also not possible to use more than once a key word even if they are in separated scopes. We faced this specific limitation when we wanted to define a name property for a Task and a State. So, we had to use the "name" key word for States and "taskName" for Tasks.

In comparison to MPS, Groovy does not give the possibility to define the grammar as precisely. It does not consider the indentations and tabulations.

6 Work Repartition

Anthony Barna:

- POC Groovy, MPS: Exception Throwing

Leo Burette:

- Groovy: Time and Parallel Extensions

Lara Defendini:

- MPS: Handling Analogical bricks and convert the report to Latex

Guillaume Savornin:

- MPS: Base scenarios + LCD screen, Time and Parallel Extensions

Anton van der Tuijn:

- Groovy: Base scenarios + Handling Analogical bricks, Exception Throwing and LCD screen extensions

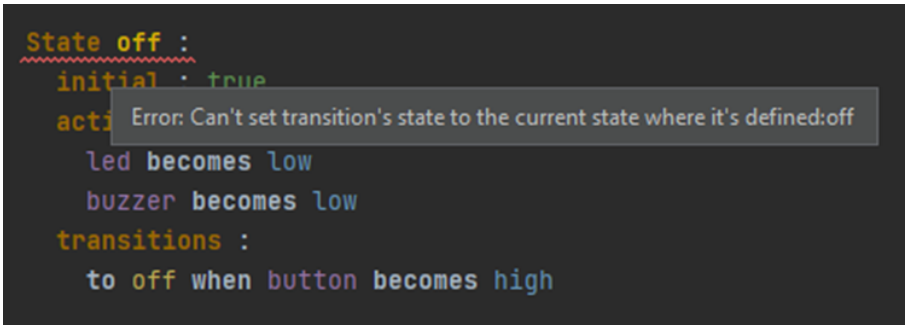
Appendices

A Convention used to write our eBNF

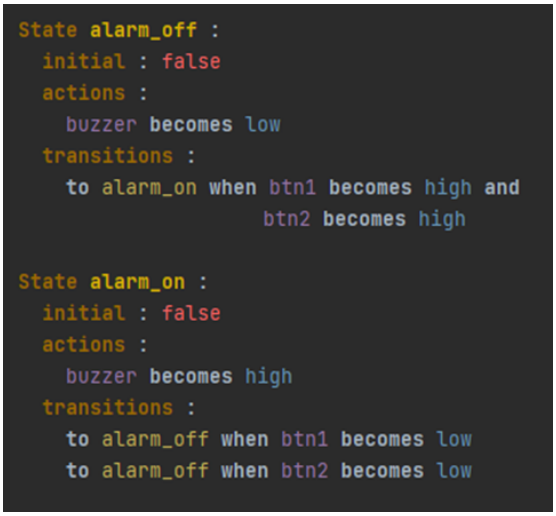
=	definition
;	termination
|	alternation
(...)	grouping
" ... "	terminal string
(...)*	zero or more
(...)+	one or more
(...)?	zero or one

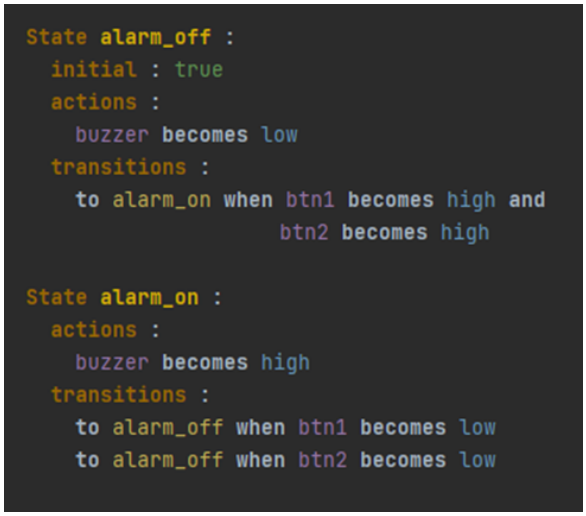
B MPS Constraints: Errors and Warnings

We added constraints to better guide the expert while he uses the language.
- In case he sets a transition to the current state, an error is displayed:

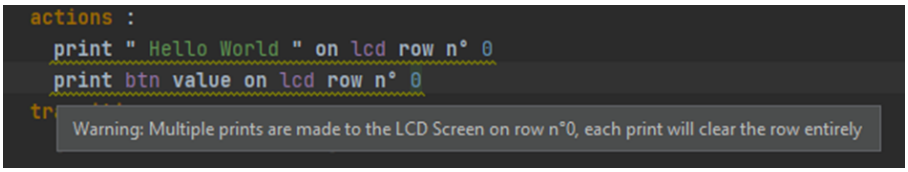


- An initial state must be selected for each Task. To make it more readable, we display the initial value to true only on the initial state, the others are hidden. In case no initial state is set yet, or all of them are set to false. They are all displayed in red with an error so that it can't compile:





- When printing on an LCD screen, if there are multiple actions on the same row, a warning is displayed to inform the expert that the last message will be overwritten:



- Multiple time transitions are allowed in the domain model, but it wouldn't make sense for the expert to have multiple ones, the longer one will never go through. So, we give a warning in case it happens: