

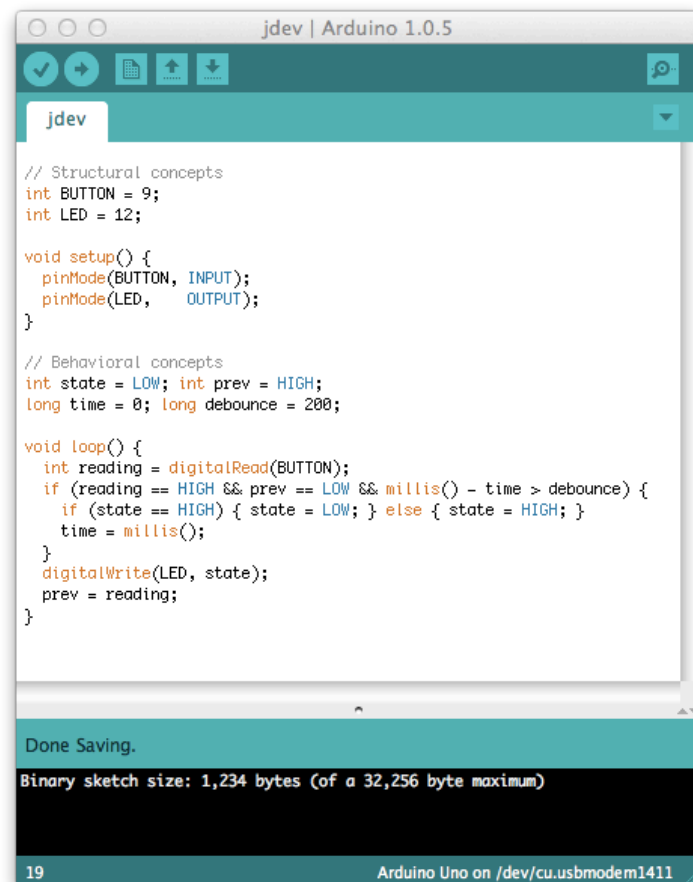
JDEV'15 – MPS Workshop

- Contact: Sébastien Mosser (mosser@i3s.unice.fr)

The objective of this lab session is to provide an overview of MPS in the context of ArduinoML. The first phase deals with the support of structural elements in our example language, and the second phase focuses on the definition of behavioral concepts.

ArduinoML

ArduinoML is a language dedicated to the modeling of pieces of software leveraging sensors and actuators on top of an Arduino microcontroller. It is specific to this class of applications, i.e., allowing anyone to model simple pieces of software that bind sensors to actuators. For example, let's consider the following scenario: "As a user, considering a button and a LED, I want to switch on the LED after pushing the button. Pushing the button again will switch it off, and so on". The essence of ArduinoML is to support the definition of such an application.



```
// Structural concepts
int BUTTON = 9;
int LED = 12;

void setup() {
  pinMode(BUTTON, INPUT);
  pinMode(LED, OUTPUT);
}

// Behavioral concepts
int state = LOW; int prev = HIGH;
long time = 0; long debounce = 200;

void loop() {
  int reading = digitalRead(BUTTON);
  if (reading == HIGH && prev == LOW && millis() - time > debounce) {
    if (state == HIGH) { state = LOW; } else { state = HIGH; }
    time = millis();
  }
  digitalWrite(LED, state);
  prev = reading;
}
```

Done Saving.

Binary sketch size: 1,234 bytes (of a 32,256 byte maximum)

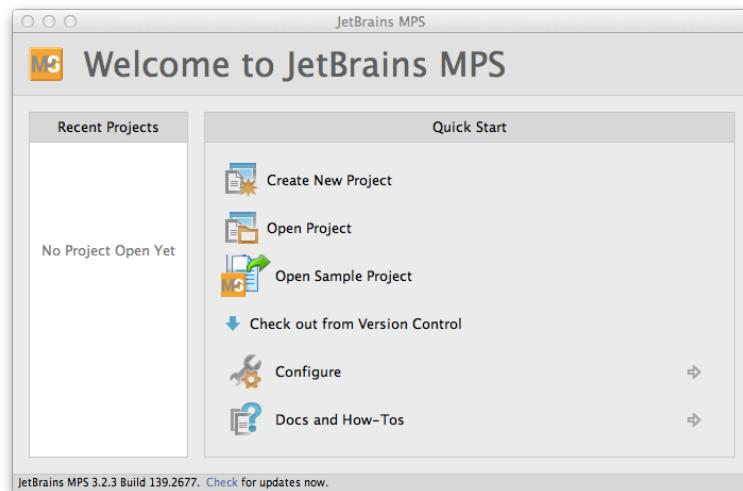
19 Arduino Uno on /dev/cu.usbmodem1411

An implementation of this application is described in the previous figure. In this “sketch”, the button is bound to pin #9, and the LED to pin #12. The infinite loop executed by the microcontroller is simple. It reads the electric signal available from the button as a digital value. If the reading is HIGH and the previous state is LOW (and considering a debounce mechanisms to avoid blinking), the state is changed according to

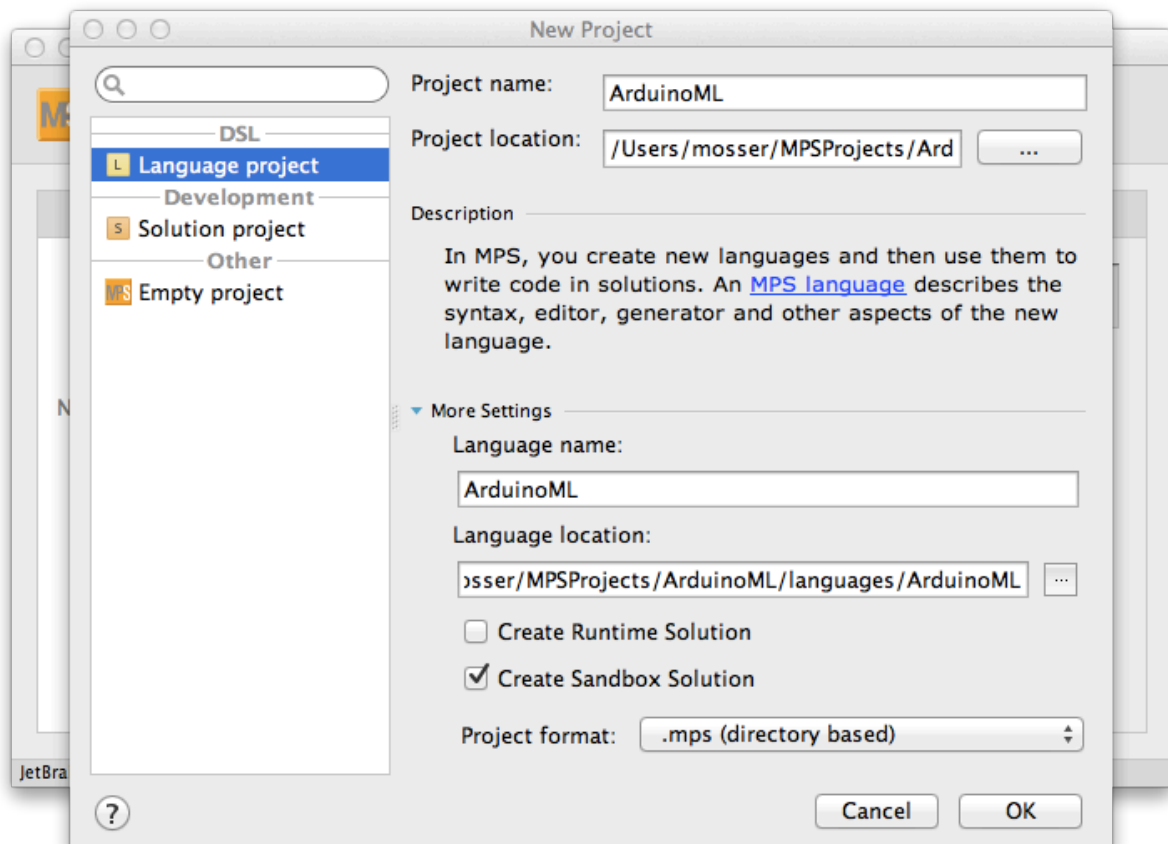
the previous one. This piece of code is not that complicated to write, but far from the business logic expressed in the previous paragraph.

Step #0: Setting up the environment

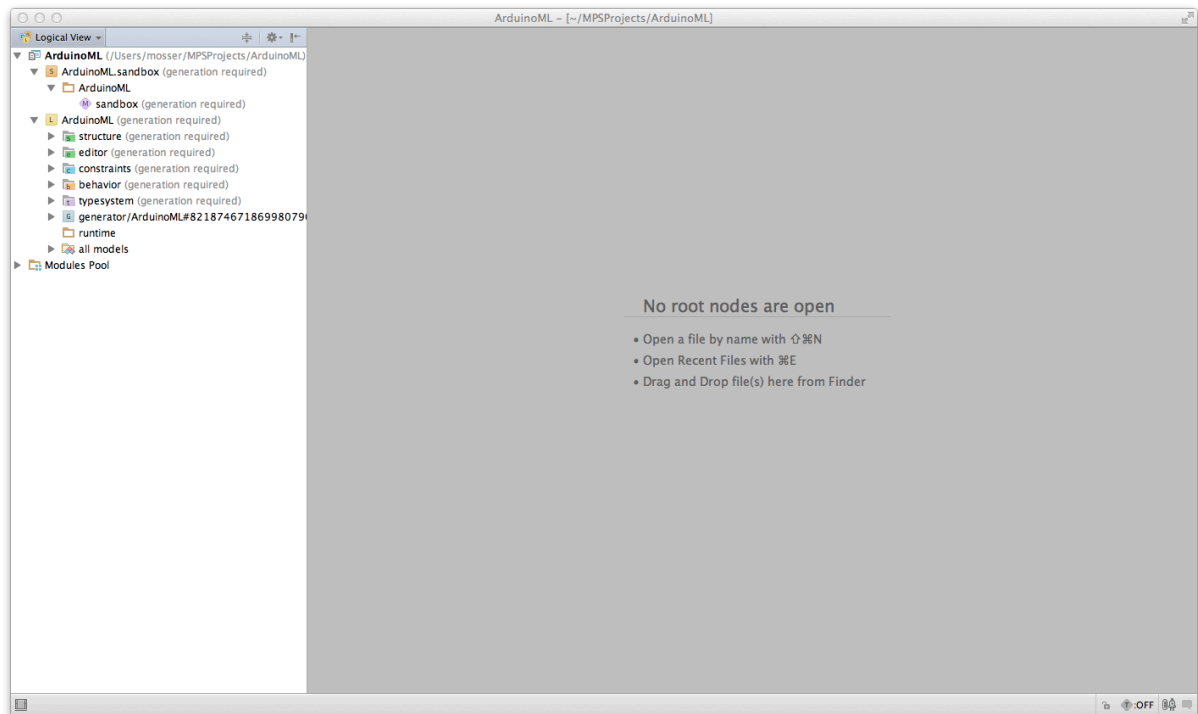
Start by creating a new project (“Create New Project”) to store our language.



The language will be named “ArduinoML”, so it is a good name for the project and the language. Tick the “Create Sandbox solution” to automatically instantiate a project dedicated to play with your language.



After indexing, the window will contain two projects. An “S”, meaning that this project is a “Solution” project, identifies the “ArduinoML.sandbox” element. It is linked to the “ArduinoML” project language, identified by an “L”.

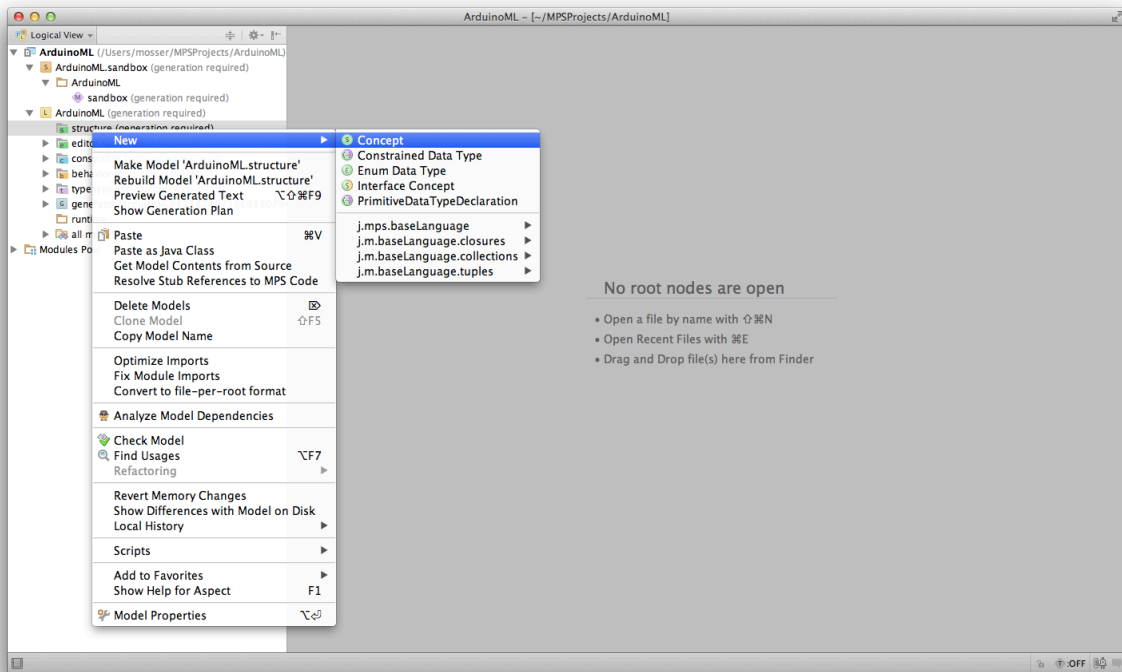


Phase #1: Handling structural concepts in ArduinoML

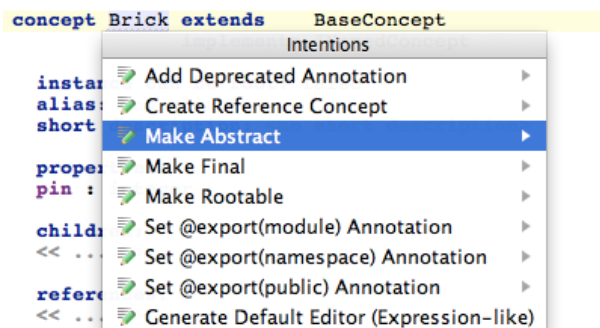
This phase focuses on the definition of language elements dedicated to the structural concepts defined in ArduinoML. Our goal here is to model the sensors and actuators bound to a given board, and generate the “void setup() {...}” function to upload on it.

Step #1.1: Creating Structural concepts for ArduinoML

The meta-model contains, for the structural description of ArduinoML applications, several concepts: Actuators, Sensors (abstracted as Bricks). These bricks are contained by an App, acting as the root of our system. Create these concepts in the “Structure” part of the language.



The Brick concept



Create a new concept, and make it abstract (Alt-enter on the name to summon the contextual menu). All bricks will have a name property, declared as a string. As meta-classes often define a “name” property, we reuse the `INamedConcept` interface provided by MPS (Ctrl-space for code completion). We add an integer property to model the pin where the brick is plugged into the Arduino board.

```
abstract concept Brick extends BaseConcept
    implements INamedConcept

instance can be root: false
alias: <no alias>
short description: <no short description>

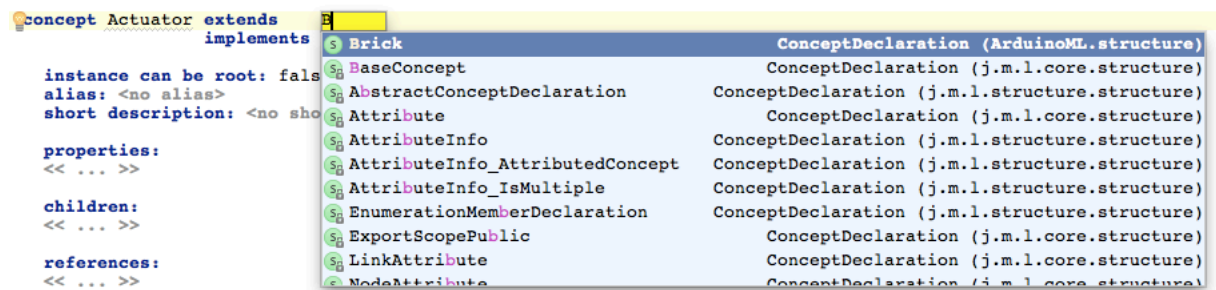
properties:
    pin : integer

children:
    << ... >>

references:
    << ... >>
```

The Sensor and Actuator concepts

These two concepts simply extend the Brick concept created in the previous paragraph. Use Ctrl-Enter for code completion.



The App concept

The App concept will contain the bricks used in the application, as children. The cardinality of this containment relationship is 1..n. An instance of App will be the root of our modeling environment. As a consequence, its property “instance can be root” is set to true.

```
concept App extends BaseConcept
implements <none>

instance can be root: true
alias: <no alias>
short description: <no short description>

properties:
<< ... >>

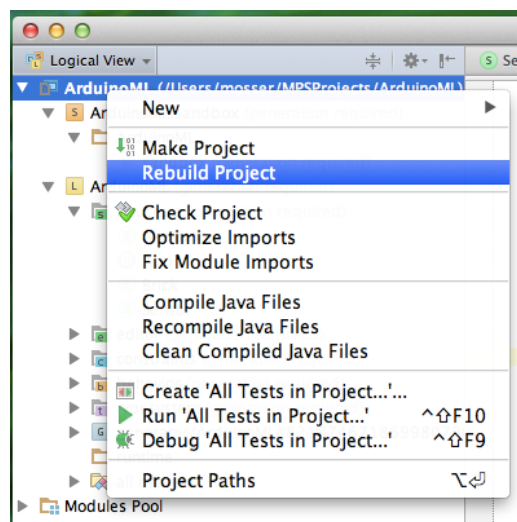
children:
bricks : Brick[1..n]

references:
<< ... >>
```

Step #1.2: Editing Models

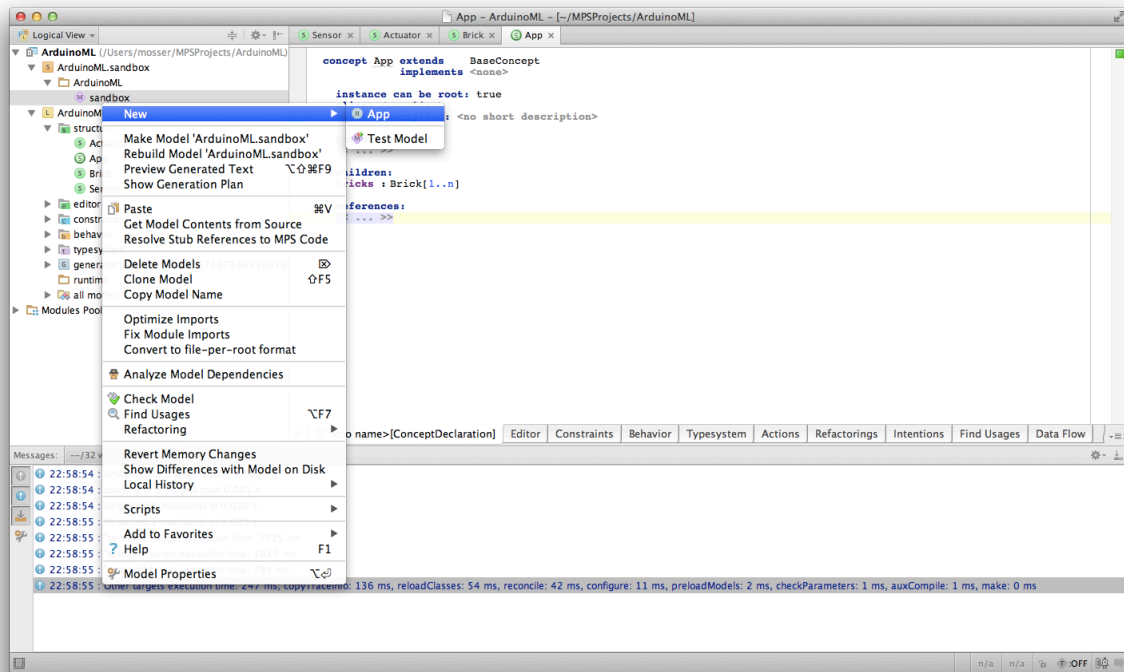
Building the system

MPS is all about “Projectional Edition”. It is time to define “projections” for our concepts. First, we will use the default editor. The preliminary step is to rebuild the project, to compile the concept modeled in the previous part, and make the concepts available in the sandbox.



Creating an App for the “Big Red Button” application

After completion of the “Rebuild” process, you can instantiate an App in the sandbox.



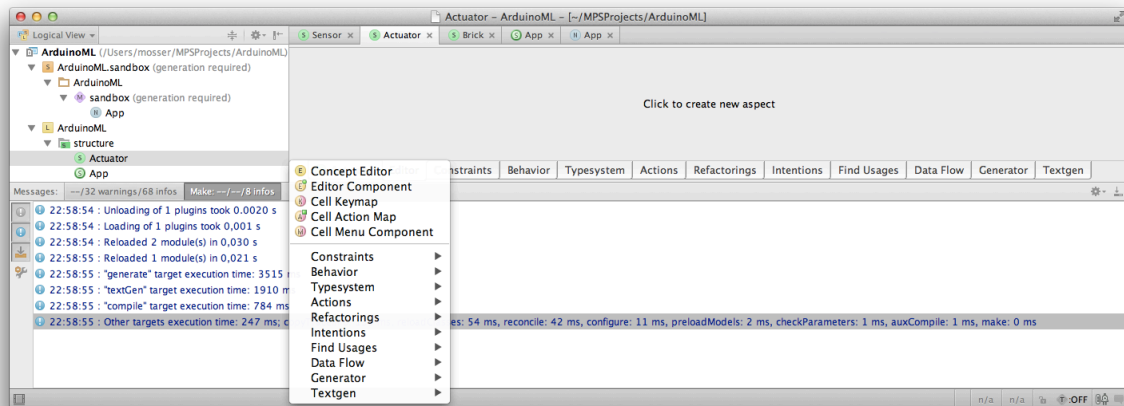
The model can be edited using code completion, using Ctrl-Enter to summon the contextual menu. Model a button (sensor) bound to pin 8, and a red led (actuator) bound to pin 12.

```
app {
    bricks :
        actuator red_led {
            pin : 12
        }
        sensor button {
            pin : 8
        }
}
```

Providing a new syntax (a new projection)

Actually, we simply filled an abstract syntax tree, through the default projection automatically provided by MPS. If you want to change the concrete syntax of your DSL, the key point is to provide another projection. The models will be automatically updated to reflect this new projection (as you are not manipulating text but projection of the AST).

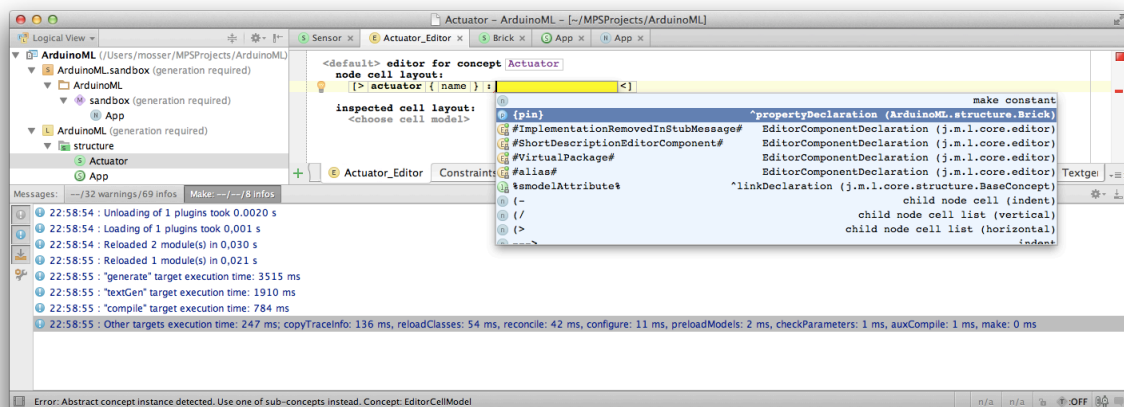
Click on the “Actuator” concept; select the “Editor” tab (bottom-left of the edition panel); click on the “+” button and select “Concept editor”.



We want to write things like “actuator red_led: 12”. This sentence is a horizontal collection of 4 elements:

“actuator” %NAME% “:” %PIN%

The syntax defined by MPS to model such a thing is to use [> ... <] to define a horizontal collection. Constants like “actuator” and “:” are defined by typing directly the expected token, and variable contents are available through code completion (Ctrl-space).



Do something similar for the Sensor concept. Rebuild the project, and open the previously created App in the Sandbox.

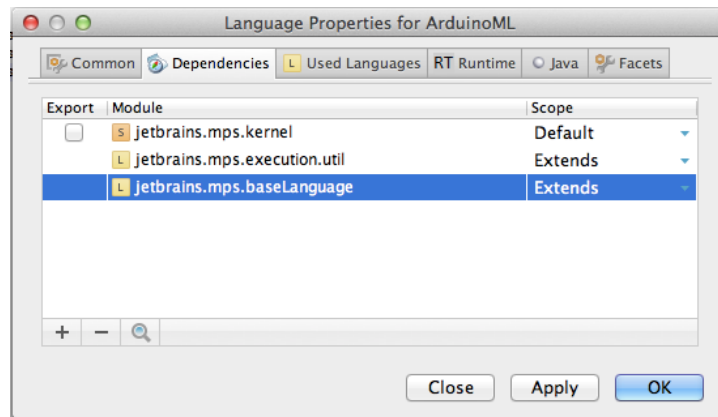
```
app {
  bricks :
    actuator red_led : 12
    sensor button : 8
}
```

Step #1.3: Generating code

MPS brings a template processor engine to generate code. Its main drawback is that this engine only works between MPS-defined languages. In this lab session, we will use a “quick and dirty” trick, by generating Java that will print the expected Arduino code. The “right” thing to do is (i) to model the Wiring language used as target or (ii) model a “text” language and uses it as a target. The process follows a map/reduce mechanism (if you are familiar with it).

Language composition: defining dependencies

Right click on the “ArduinoML” language, select “Module Properties” and the “Dependencies”. Our language needs the MPS kernel for code generation purpose. We also extends the “execution.utils” and “baseLanguage” languages for expressiveness purpose.



Making the App concept executable

Edit the “App” concept, in order to implement the IMainClass interface (provided by the newly declared dependencies).

```
concept App extends BaseConcept
            implements IMainClass

instance can be root: true
alias: <no alias>
short description: <no short description>

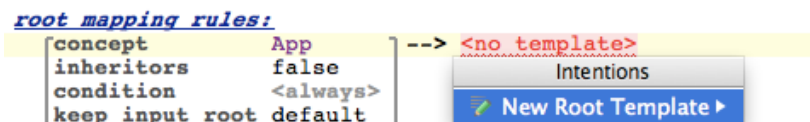
properties:
<< ... >>

children:
bricks : Brick[1..n]

references:
<< ... >>
```

Create the Root Mapping rule

Press “Enter” under the “Root mapping rule” element in the main@generator template definition. Use Ctrl-space to select the “App” concept as input of the rule. We will map an App to a Class. On the left part of the rule, use Alt-enter to summon the contextual menu and select “New root Template”.



You can now edit the target template through a projectionnal editor dedicated to Java.

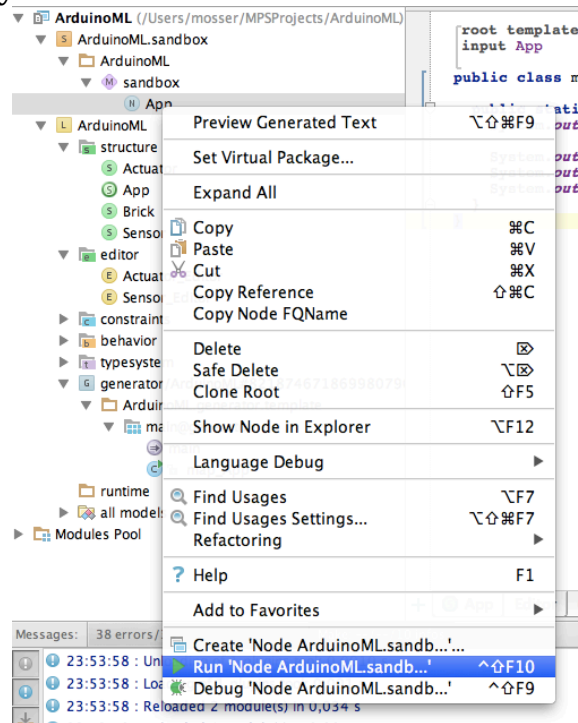
```
[root template]
input App

public class map_App {

    public static void main(string[] args) {
        System.out.println("// Code Generated by ArduinoML");

        System.out.println("void setup() {");
        System.out.println(" // Here comes brick declaration");
        System.out.println("}");
    }
}
```


Rebuild the projects, and then right-click on the previously modeled App. The result of the execution is displayed on the console.



Concretely generating pin declaration instructions with Reduction rules

Sensor and Actuators will use different generators. Such elements are called “reduction rules”. We start by defining a reduction rule for Sensor.

Create a new reduction rule following the same steps than the root mapping one, excepting that you’ll define it under “Reduction rules”.

```
reduction rules:
[concept Sensor] --> reduce Sensor
[inheritors false]
[condition <always>]
```

The reduction rule contains a BlockStatement (Alt-enter for code completion)

```
template reduce_Sensor
input Sensor

parameters
<< ... >>

content node:
blockS
Replace with instance of BlockStatement concept
```

The code to be generated is the following: “pinMode(“ + pinNumber + “,INPUT);”. Thus, we put such a statement inside the code block. As the template engine will process this code, we wrap it inside a “template fragment” (alt-enter at the end of the statement). The template fragment models the text that will be returned from the reduction rule to the caller one. Rebuild the project of necessary, to remove an error if MPS is not able to recognize the template fragment.

```

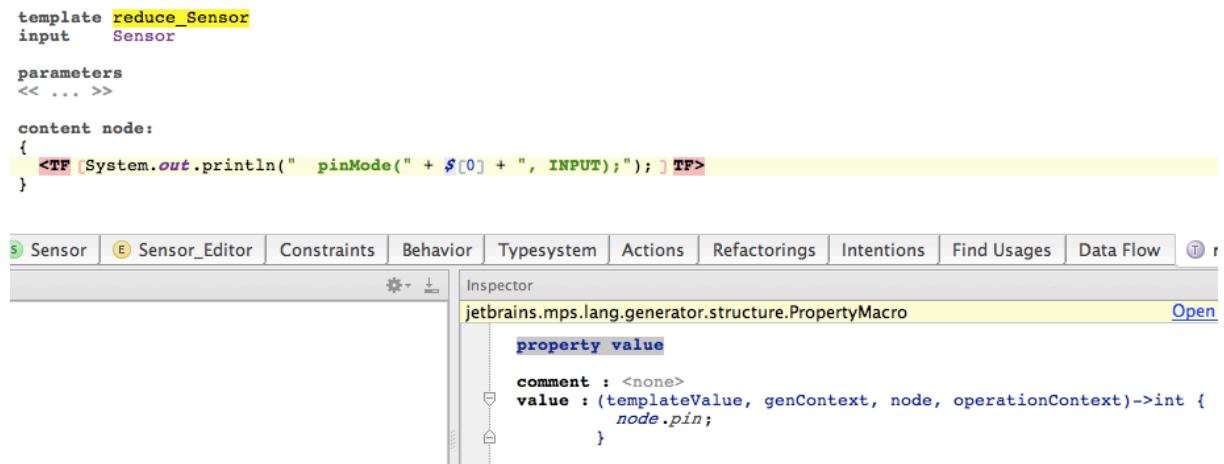
template reduce_Sensor
input Sensor

parameters
<< ... >>

content node:
{
  <TF [System.out.println(" pinMode(" + 0 + ", INPUT);"); ] TF>
}

```

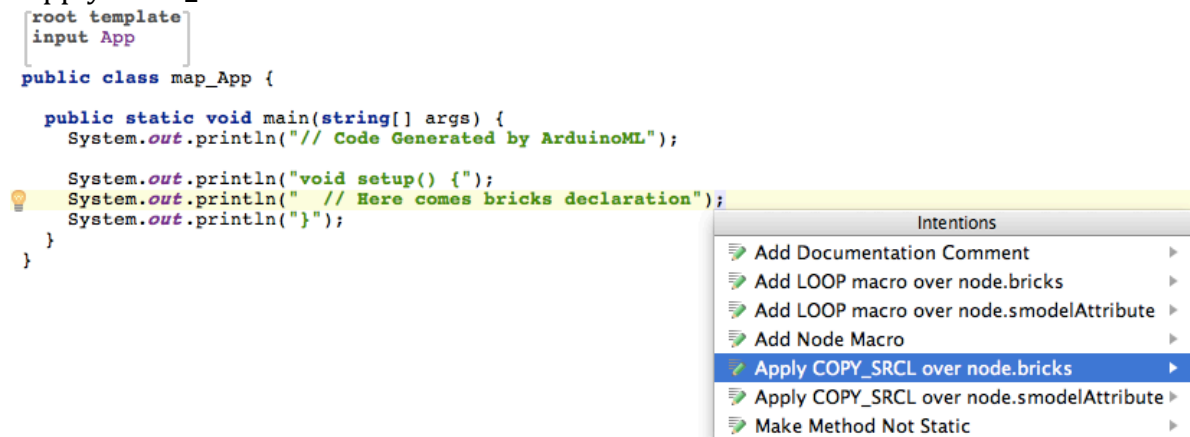
We use 0 as a placeholder “pinNumber”, to be replaced by the real one. Use Alt-enter on this number to summon the contextual menu. Select “Add property macro: node.pin”



Do something very similar for Actuators, replacing INPUT by OUTPUT.

Calling the reduction rules from the main mapping one

Actually, you just want to replace the static declaration stating that “Here comes the bricks definition” by the real ones. Use alt-Enter at the end of this statement, and select “Apply COPY_SRCL over node.bricks”. That’s all folks



Rebuild the project, and re-run the App model. Congratulations, the setup phase is now supported by your implementation of ArduinoML!

```

Run Node ArduinoML.sandbox.map_App

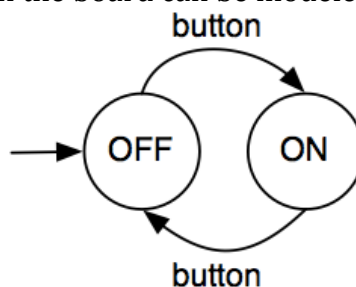
/System/Library/Java/JavaVirtualMachines/1.6.0.;
// Code Generated by ArduinoML
void setup() {
  pinMode(12, OUTPUT);
  pinMode(9, INPUT);
}

Process finished with exit code 0

```

Phase #2: Supporting behavioral concepts

The behavior to be uploaded on the board can be modeled as an automaton.



Step #2.1: Adding the concepts necessary to model ArduinoML behaviors

To support this behavior, we need to model the following concepts in our meta-model:

- A State is a named entities, containing a lists of “actions” and a transition:
 - An Action sends a “signal” to a referenced actuator
 - A Transition is linked to a referenced sensor, triggered by a given value and targeting a given state
- The sensed values are defined as an enumerated type, in “low” and “high”.
- The App concepts must be edited to contain now a list of States, and a reference to the initial one. We also add a name to the App, so it is more “business oriented”.

```

concept State extends <default>
  implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
    << ... >>

  children:
    actions : Action[0..n]
    transition : Transition[1]

  references:
    << ... >>

  enumeration datatype STATUS
    member type : boolean
    no default : true
    null text : <none>
    member identifier : derive from presentation

    value true presentation high
    value false presentation low

```

```

concept Transition extends BaseConcept
  implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
    status : STATUS

  children:
    << ... >>

  references:
    sensor : Sensor[1]
    target : State[1]

concept Action extends BaseConcept
  implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
    status : STATUS

  children:
    << ... >>

  references:
    actuator : Actuator[1]

concept App extends BaseConcept
  implements IMainClass
  INamedConcept

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
    << ... >>

  children:
    bricks : Brick[1..n]
    state : State[1..n]

  references:
    init_state : State[1]

```

Rebuild the whole system (language + solution). The solution will generate errors, as the previously defined models are not compliant with the new meta-model.

Step #2.2: Model the “red button” example using the default editor

Edit the previously defined model with the default editor. We need to add 2 states, and transitions from one state to the other.

```

app RedButton init_state : off {
  state :
    state off {
      transitions :
        transition sensor : button target : on {
          status : high
        }
      actions :
        action actuator : red_led {
          status : low
        }
    }
  state on {
    transitions :
      transition sensor : button target : off {
        status : high
      }
    actions :
      action actuator : red_led {
        status : high
      }
    }
  bricks :
    actuator red_led : 12
    sensor button : 9

```

The default editor is definitively not suited for our needs!

Step #2.3: Providing a better editor

We need to define better projections for Actions, Transitions and State (we'll keep the default projection for App for the sake of simplicity).

Referenced nodes (e.g., in an Action, the actuator to be modified is references) are defined as ... references! One must specify the attribute to be used to identify the referenced element among the defined ones. We'll use here the name of the actuator. The situation is very similar for Transition. Using the given projection, one is able to model an Action and a Transition as the following:

- Action: *actuator_name* <= *expected_value*
- Transition: *sensor_name* is *expected_value* => *target_state_name*

```
<default> editor for concept Action
node cell layout:
[- ( (% actuator % -> { name } ) <= { status } ) -]

inspected cell layout:
<choose cell model>

<default> editor for concept Transition
node cell layout:
[- ( (% sensor % -> { name } ) is { status } => ( (% target % -> { name } ) ) -) ]

inspected cell layout:
<choose cell model>
```

The State projection will basically define the name of the state, and provide a vertical list of actions, followed by transitions. We use the ---> keyword to specify indentation, and the (/ operator to specify that actions should be displayed vertically.

```
<default> editor for concept State
node cell layout:
[/
[- { name } { -}
[- ---> (/ % actions % /) -]
[- ---> % transition % -]
]
/]

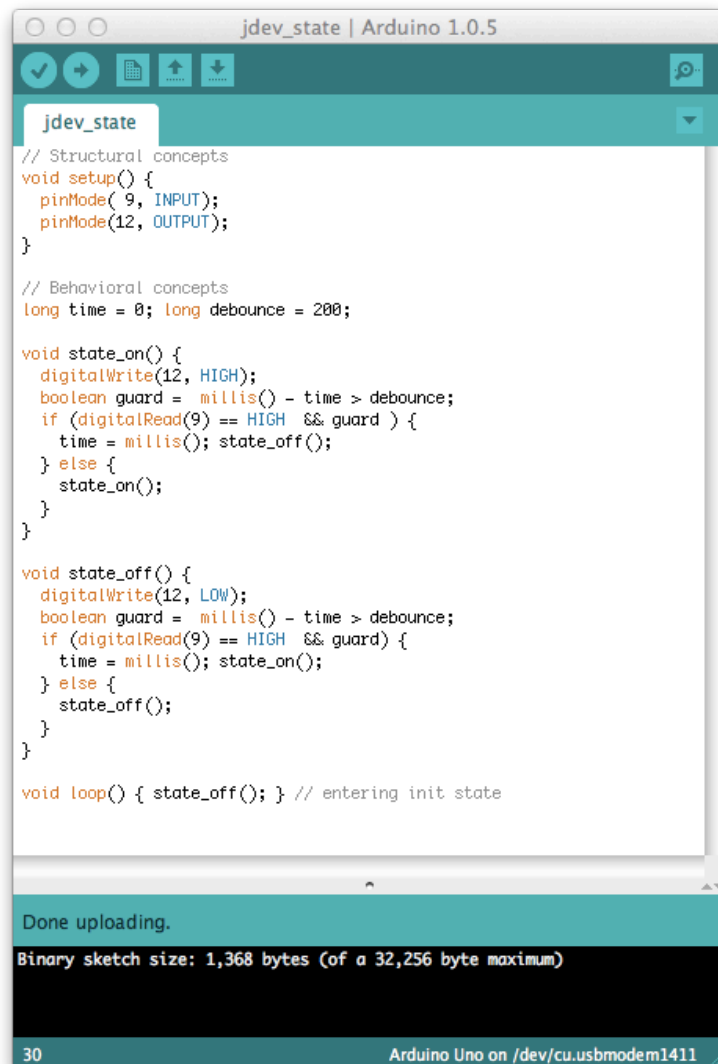
inspected cell layout:
<choose cell model>
```

With these projection defined and after a rebuild of the project, open the previously defined model. As we are working with projection, your new syntax is immediately available!

```
app RedButton init_state : off {
  states :
    off {
      red_led <= low
      button is high => on
    }
    on {
      red_led <= high
      button is high => off
    }
  bricks :
    actuator red_led : 12
    sensor button : 9
}
```

Step #3.3: Generating behavioral code

To ease the generation of the code to be executed on the board, we propose here a simple refactoring of the initial implementation depicted at the beginning of this workshop description.



```
jdev_state | Arduino 1.0.5

jdev_state
// Structural concepts
void setup() {
  pinMode( 9, INPUT);
  pinMode(12, OUTPUT);
}

// Behavioral concepts
long time = 0; long debounce = 200;

void state_on() {
  digitalWrite(12, HIGH);
  boolean guard = millis() - time > debounce;
  if (digitalRead(9) == HIGH && guard ) {
    time = millis(); state_off();
  } else {
    state_on();
  }
}

void state_off() {
  digitalWrite(12, LOW);
  boolean guard = millis() - time > debounce;
  if (digitalRead(9) == HIGH && guard ) {
    time = millis(); state_on();
  } else {
    state_off();
  }
}

void loop() { state_off(); } // entering init state

Done uploading.
Binary sketch size: 1,368 bytes (of a 32,256 byte maximum)

30 Arduino Uno on /dev/cu.usbmodem1411
```

We consider here a state-based implementation, where states are implemented as function and transitions as function calls. Each state starts by executing its actions, and then a transition mechanism implementing the “debounce” pattern is generated (the “guard” is used to reduce noise).

Editing the App mapping template

First, we edit the App root template to add the needed placeholders in the main java class. We need a place where the different state functions will be generated, and to initialize the contents of the loop function with a call to the initial state function.

```

[template
input App
]
public class map_App {

    public static void main(string[] args) {
        System.out.println("// Code Generated by ArduinoML\n");

        System.out.println("// Structural concepts");
        System.out.println("void setup() {");
        $COPY_SRCL$(System.out.println(" // Here comes bricks declarations"); )
        System.out.println("}");

        System.out.println("\n// Behavioral concepts");
        System.out.println("long time = 0; long debounce = 200;\n");

        System.out.println(" // Here comes states declarations");

        System.out.println("\nvoid loop() { state_ + "init_state" + "() ; } // Entering init state\n");
    }
}

```

Calling the initial state

The easy placeholder is the “init_state” string in the “loop” function, to replace as a property macro. It will be bound to the name of the initial state. As this is a transitive relation, MPS will not provide an automatic way to add it. Add a “property macro”, and edit its contents to refer to the good value.

```

property value

comment : <none>
value : (templateValue, genContext, node, operationContext)->string {
    node.init_state.name;
}

```

Reducing Actions

We then create a reduction rule dedicated to the “Action” concept. Its implementation is straightforward, using an integer placeholder bound to “node.actuator.pin” and a signal placeholder bound to the enumeration value.

```

template reduce_Action
input Action

parameters
<< ... >>

content node:
{
    <TF [System.out.println(" digitalWrite(" + $[0] + ", " + "$[SIGNAL]" + ");"); ] TF>
}

```

For the latter, the Enum is a Boolean type, and the template expects a String. One can use a ternary form to handle this issue.

```

property value

comment : <none>
value : (templateValue, genContext, node, operationContext)->string {
    node.status ? "HIGH" : "LOW";
}

```

Reducing transitions

The template is quite straightforward to write too, based on the executable code described previously

```

template reduce_Transition
input Transition

parameters
<< ... >>

content node:
<TF {
    System.out.println(" if (digitalRead(" + $[0] + ") == " + "$[SIGNAL]" + " && guard ) {");
    System.out.println("     time = millis(); state_ + "$[STATE]" + "() ; ");
    System.out.println(" } else { state_ + "$[STATE_NAME]" + "() ; }");
} TF>

```

The main issue here is the “STATE_NAME” to be used in the “else” condition. This state is only known by the caller of the reduction rule, and cannot be known at the level of this

reduction rule. We assume that this value is present in the generation context, using a session object named "current_state_name".

```
property value

comment : <none>
value : (templateValue, genContext, node, operationContext)->string {
    "" + genContext.session object [ "current_state_name" ];
}
```

Reducing states

```
template reduce_State
input State

parameters
<< ... >>

content node:
<TF> {
    System.out.println("void state_" + "${STATE_NAME}" + "() {}");
    $COPY_SRC$ [System.out.println(" // Here comes the actions"); ]
    System.out.println(" boolean guard = millis() - time > debounce;");
    $COPY_SRC$ [System.out.println(" // Here comes the transition"); ]
    System.out.println("}\n");
} <TF>
```

In the call to the reduction rule associated to transition, we fill the session with the current name of the state before calling the reduction rule on "node.transition"

```
copy/reduce node

comment : <none>
mapping label : <no label>
mapped node : (genContext, node, operationContext)->node<> {
    genContext.session object [ "current_state_name" ] = node.name;
    node.transition;
}
```

Generating ArduinoML code

Rebuild the project, and run the previously modeled application. That's all folks!

```
Run Node RedButton
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java -classpath /Users/moss
// Code Generated by ArduinoML

// Structural concepts
void setup() {
    pinMode(12, OUTPUT);
    pinMode(9, INPUT);
}

// Behavioral concepts
long time = 0; long debounce = 200;

void state_off() {
    digitalWrite(12, LOW);
    boolean guard = millis() - time > debounce;
    if (digitalRead(9) == HIGH && guard ) {
        time = millis(); state_on();
    } else { state_off(); }
}

void state_on() {
    digitalWrite(12, HIGH);
    boolean guard = millis() - time > debounce;
    if (digitalRead(9) == HIGH && guard ) {
        time = millis(); state_off();
    } else { state_on(); }
}

void loop() { state_off(); } // Entering init state

Process finished with exit code 0
```