

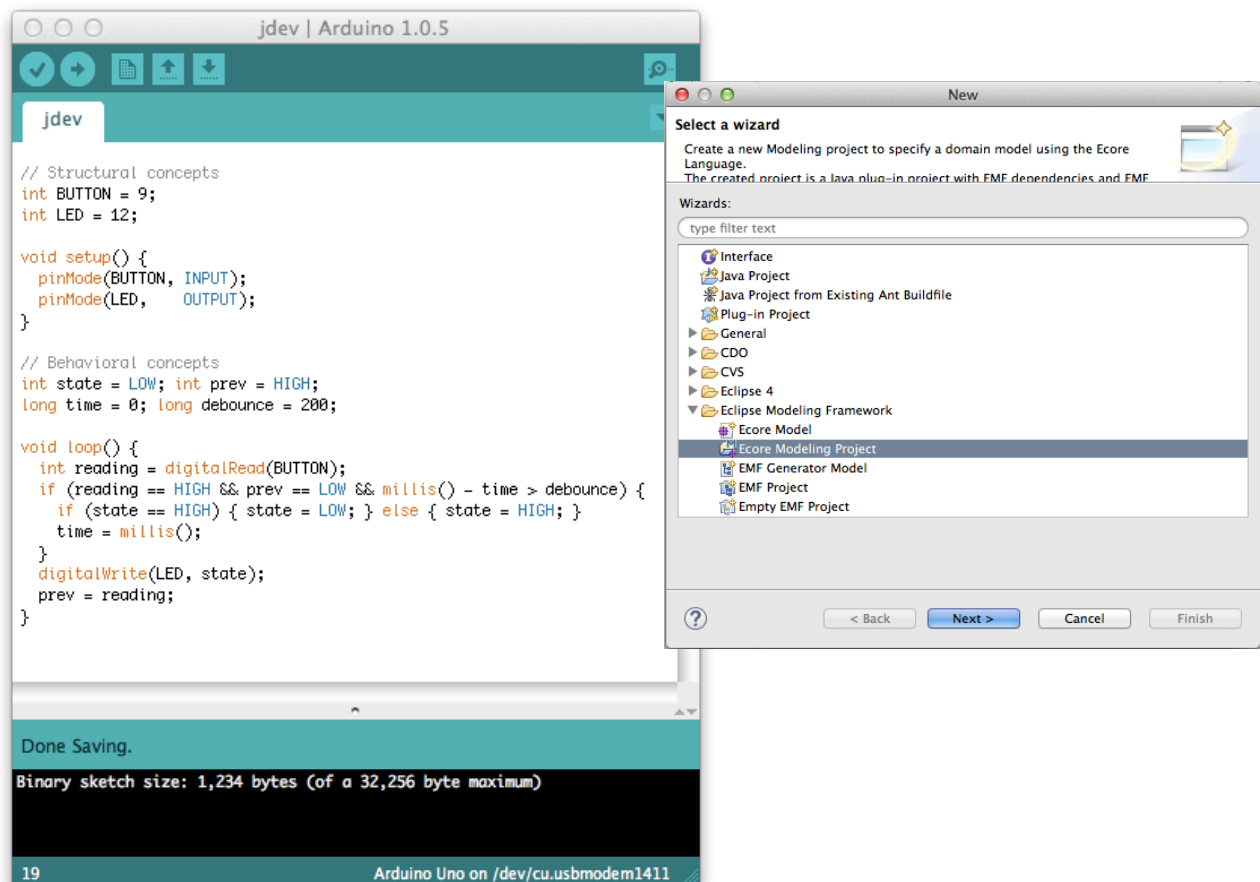
DSL – EMF-Xtext Workshop

Contact: Sébastien Mosser (mosser@i3s.unice.fr) – Ivan Logre (logre@i3s.unice.fr)

The objective of this lab session is to provide an overview of DSL design using the Eclipse EMF-Xtext stack 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. The design of the meta-classes of our example domain uses Eclipse Modeling Framework, and design of the concrete syntax to instantiate models conform to this meta-model uses Xtext.

Reminder on 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.



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

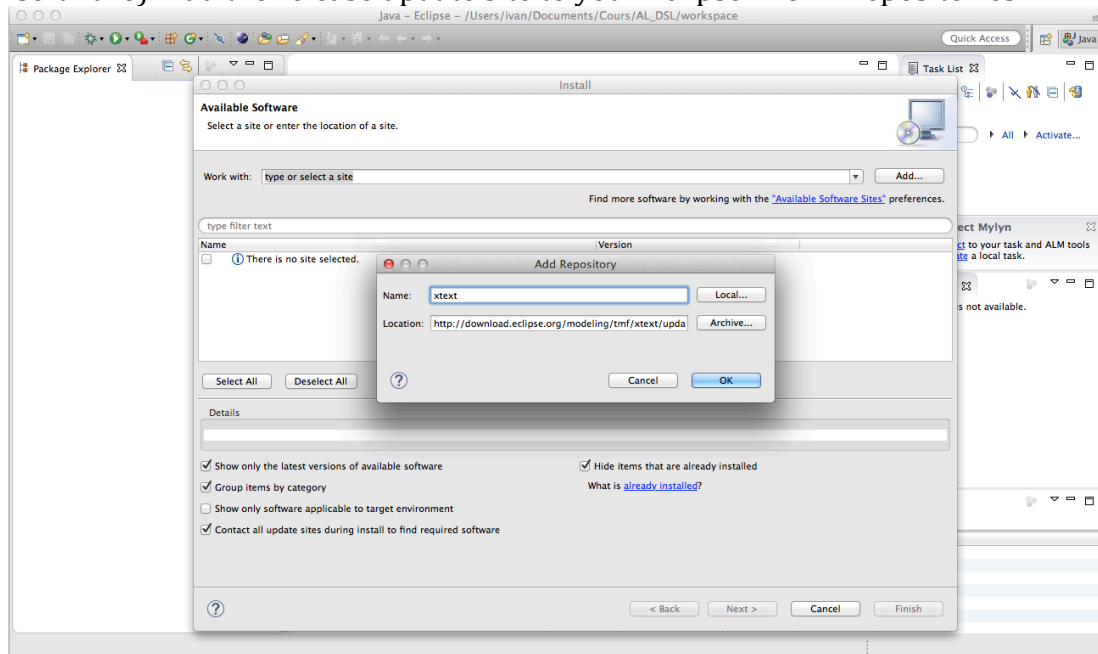
button as a digital value. If the reading is HIGH and the previous state is LOW (and considering a debounce mechanisms to avoid blinking), the state is changed according to the previous one. This piece of code is not that complicated to write, but far from the business logic expressed in the previous paragraph.

Step #0: Setting up the environment

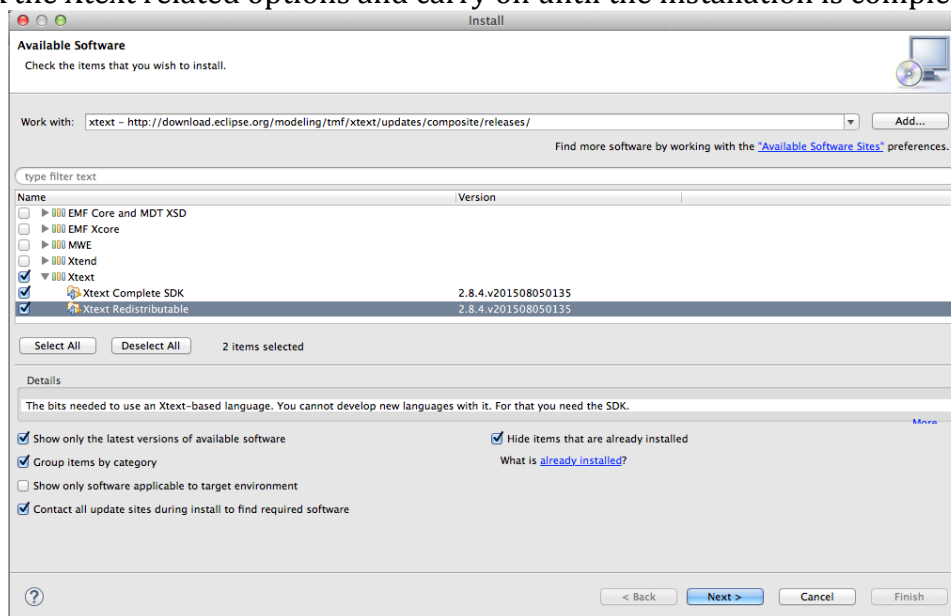
First, we need to download an Eclipse IDE with pre-installed EMF from:

<http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/mars1>

Start Eclipse, and install the Xtext Plugin [using the Install Manager](#) (Menu Help > Install new software). Add the release update site to your Eclipse known repositories



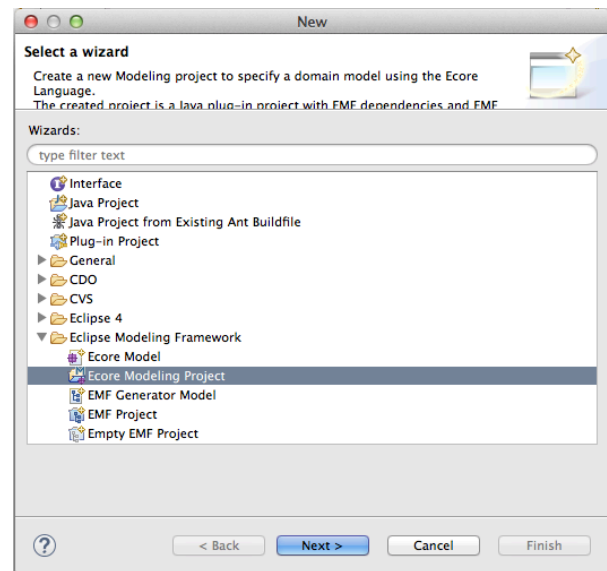
Then tick the Xtext related options and carry on until the installation is complete.



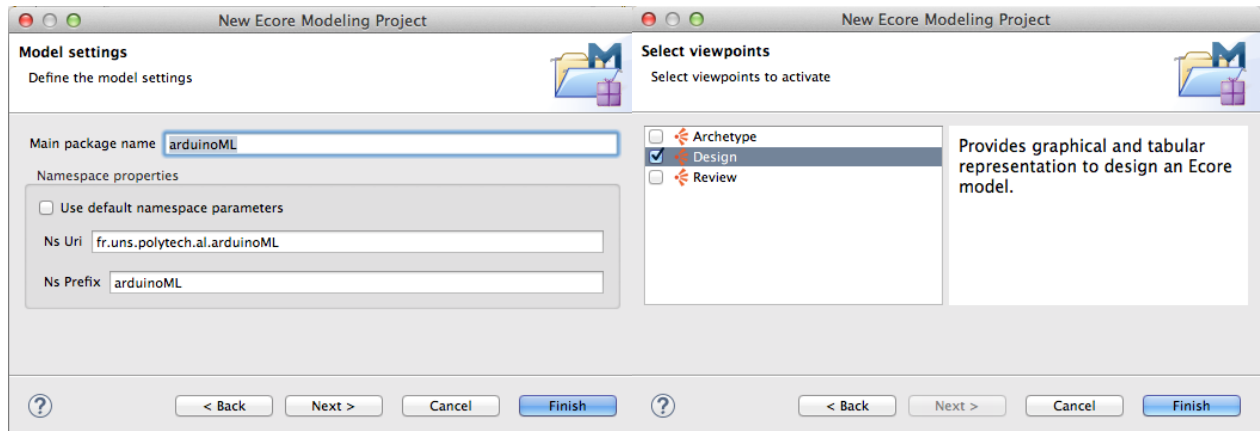
Create a new project

Right click on the explorer panel (on the left), and create a new “Ecore Modeling Project”. The language will be named “ArduinoML”, so it is a good name for the project and the language. Choose a relevant workspace for this project in your file system, e.g. a folder set up for your favorite versioning control system.

The “main package name” will be the default name of all our modeling files, so “arduinoML” is a good choice. Choose a Ns Uri and Prefix to remember for, e.g. ‘fr.uns.polytech.al.arduinoML’ & ‘arduinoML’.



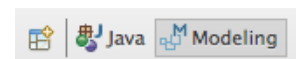
Keep only the ‘Design’ checkbox and finish.



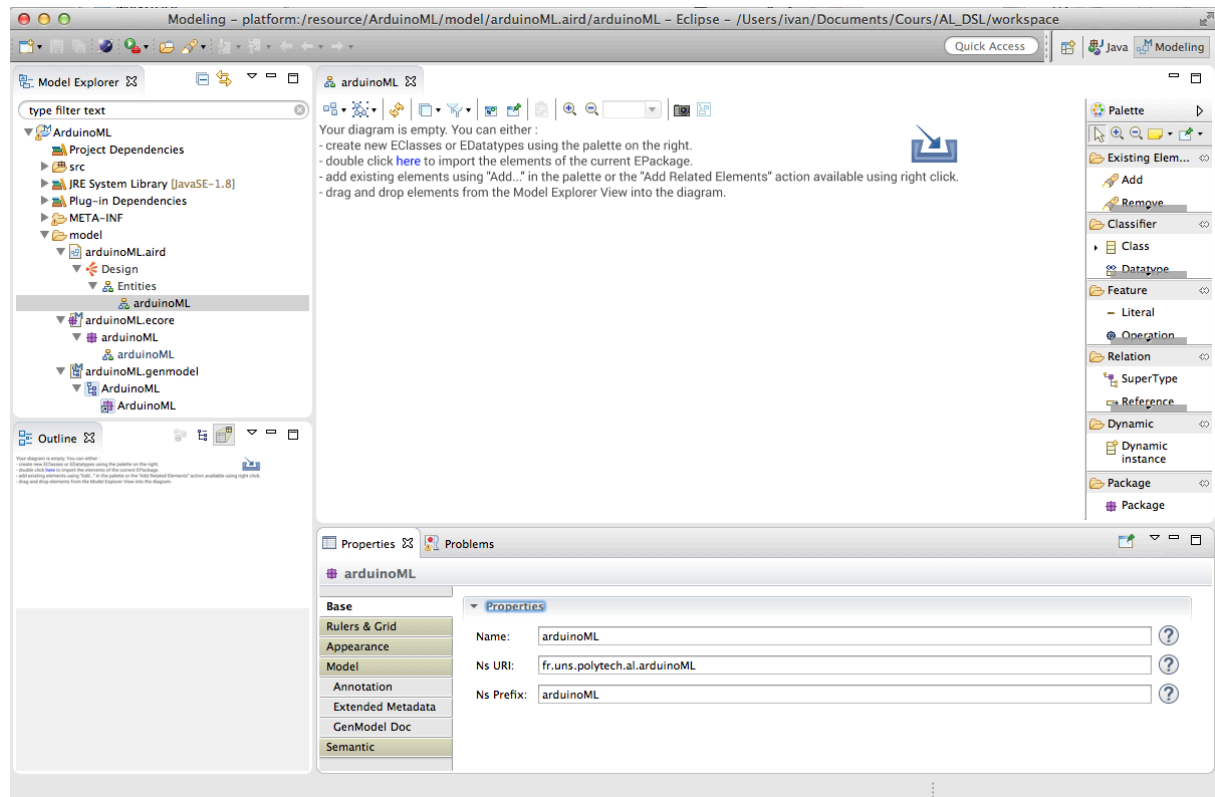
EFM generate three files:

- the .ecore file contains the structural information of our meta model (its concepts and their relations and attributes)
- the .aird file helps to edit the .ecore, we can open the graphical editor from here
- the .genmodel file allows the generation of assets of code corresponding to our meta-model

Be sure to use the ‘Modeling’ perspective of Eclipse to access relevant panels and options for this project (top-right of the window).



At this point you should get the same set up that the next screenshot.

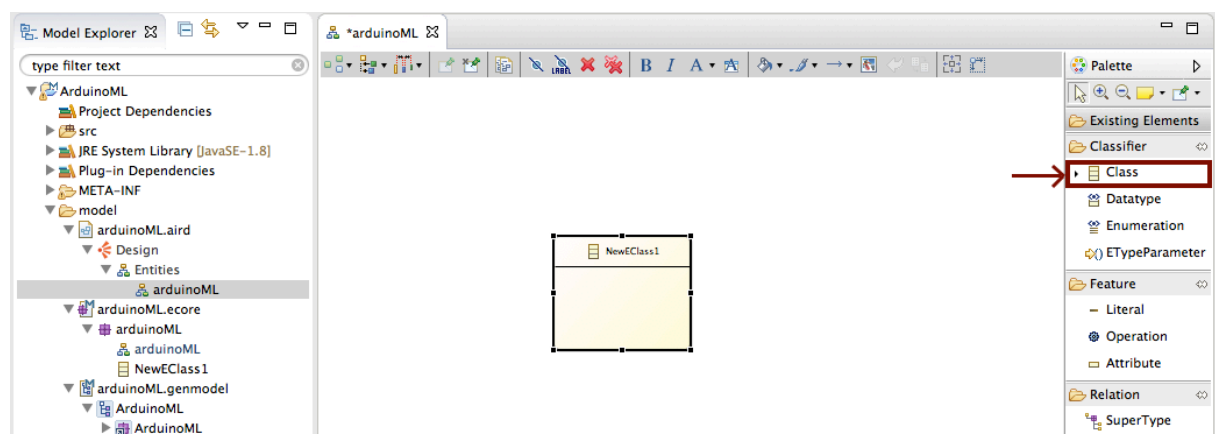


Phase #1: Handling structural meta elements in ArduinoML

This phase focuses on the definition of language abstract elements dedicated to the structural concepts defined in ArduinoML. Our goal here is to model the sensors and actuators bound to a given board.

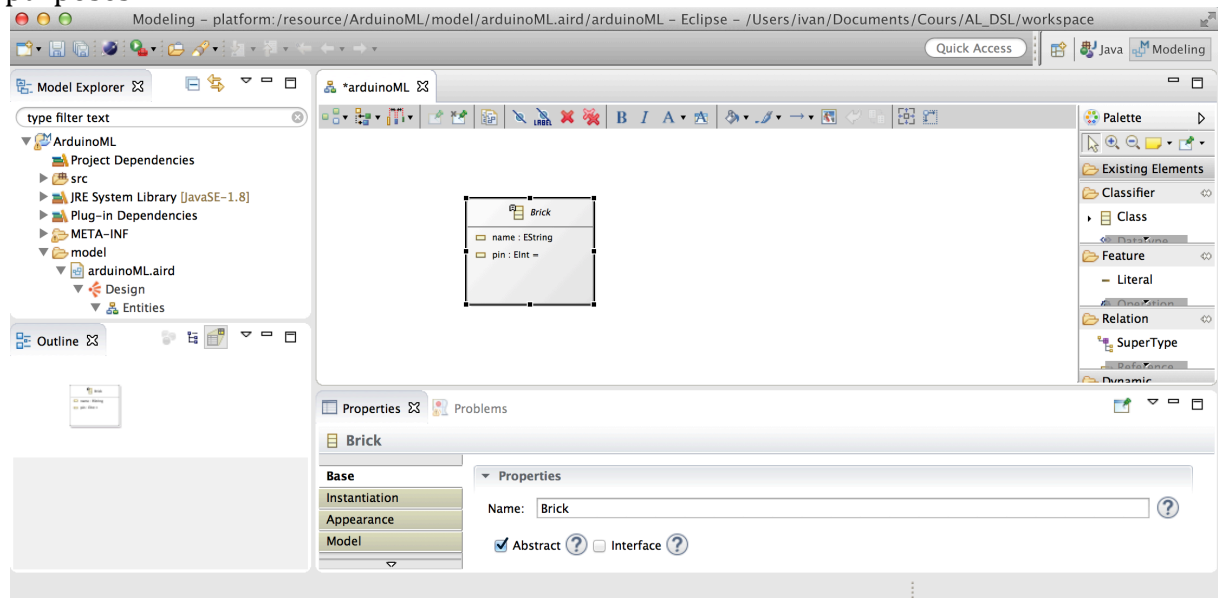
Step #1.1: Creating Structural concepts for ArduinoML

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. We are going to create graphically these meta elements using EClasses.



The Brick meta class

Create a new EClass, and make it abstract (through the Properties view). All bricks will have a name attribute, declared as a string, add it using drag and drop from the Palette on the right or by using the pop-up helper on the brick graphical representation. We add an integer attribute to model the pin where the brick is plugged into the Arduino board. Note that EMF offers to use EString and EInt instead of String and Int for generation purposes.

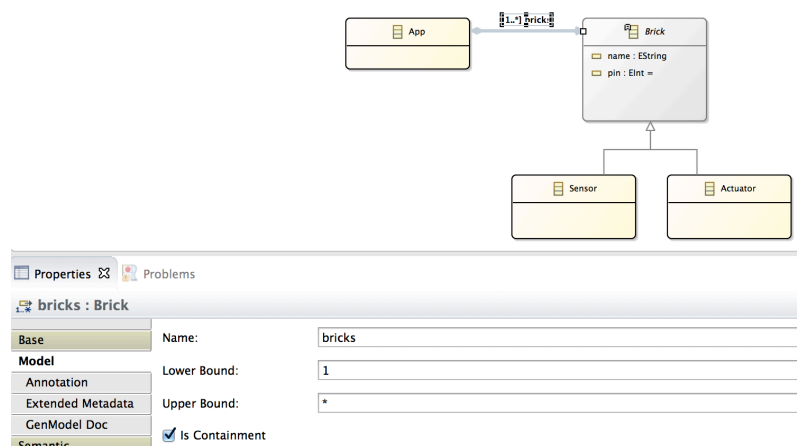


The Sensor and Actuator concepts

These two meta classes simply extend the Brick concept created in the previous paragraph. Connect them with a 'SuperType' relation.

The App concept

The App meta class will contain the bricks used in the application, as children. The cardinality of this composition relationship is 1..* (bounds). An instance of App will be the root of our modeling environment. It means that all the others concepts must be contained (even transitively) by it.



Step #1.2: Instantiate Models

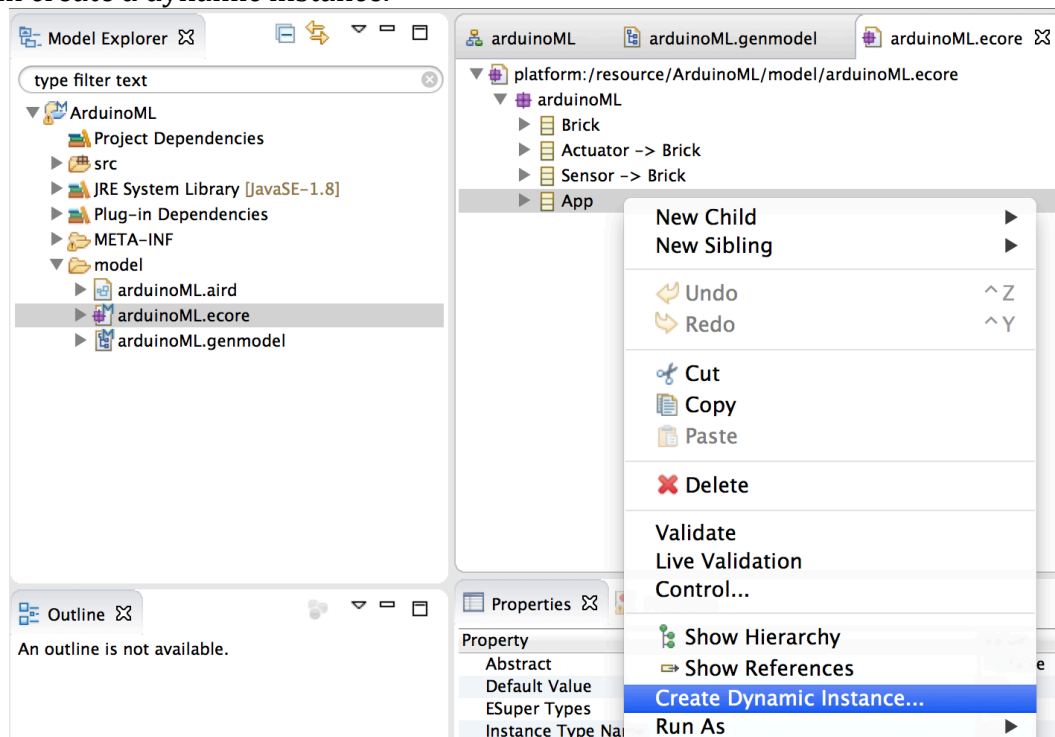
Dynamic .xmi instances

EMF provides an XML base default editor to enrich our meta model (which is not very useful in comparison with the graphical editor we just used). Note that both representations are synchronized as they work concurrently on the same xml file storing our domain, so be careful with conflicting modifications.

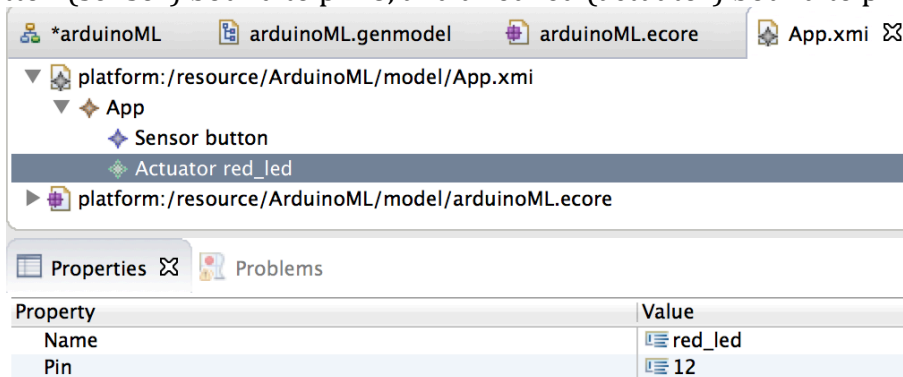
It also permits us to instantiate models conform to the meta model we defined. It is useful to take the time checking that our structural definition of the domain allows us to design meaningful instances.

Creating an App for the “Big Red Button” application

By opening our ‘.ecore’ file with the Sample Ecore Model Editor, we reach a tree representation of our structural domain. By right-clicking the root meta element ‘App’, we can create a dynamic instance.

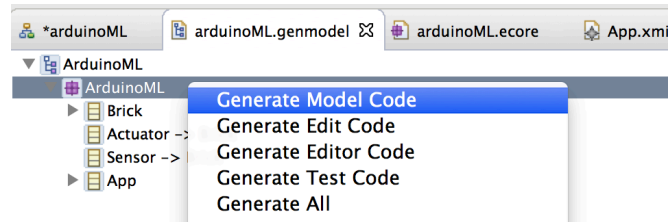


Model a button (sensor) bound to pin 8, and a red led (actuator) bound to pin 12.

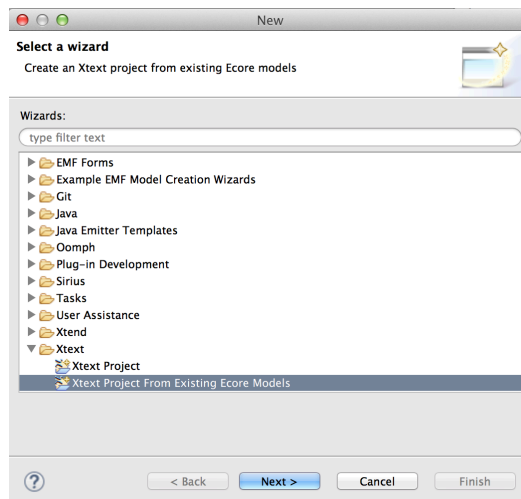


Default concrete syntax via Xtext

From the .genmodel files, you can generate Java code. Right-click on ArduinoML and select 'Generate Model Code'.

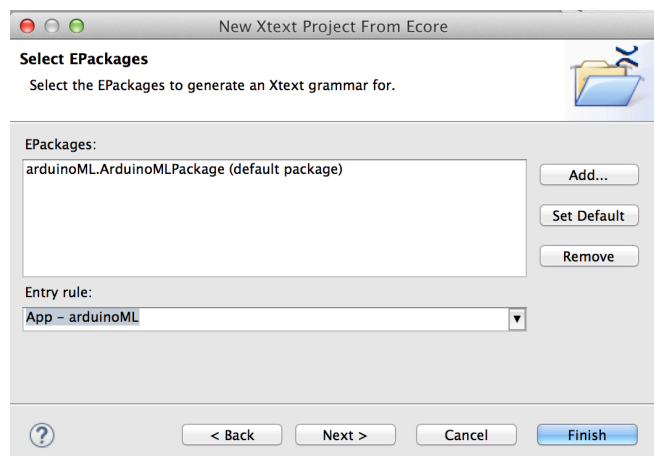
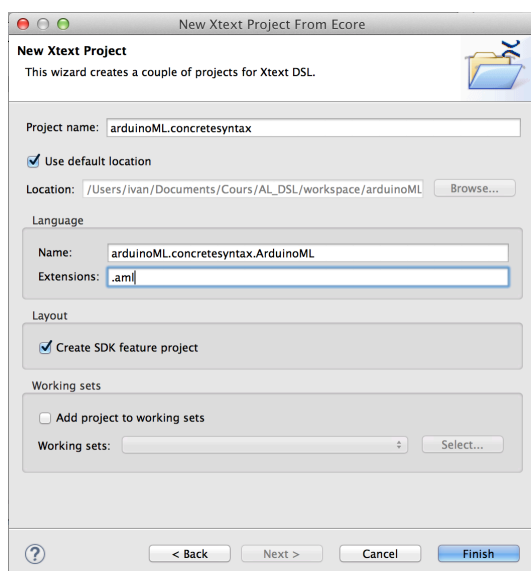


This creates the Java implementation of the EMF model in the current project in the src folder of your current project. This Java Code is needed by Xtext to generate a default syntax.

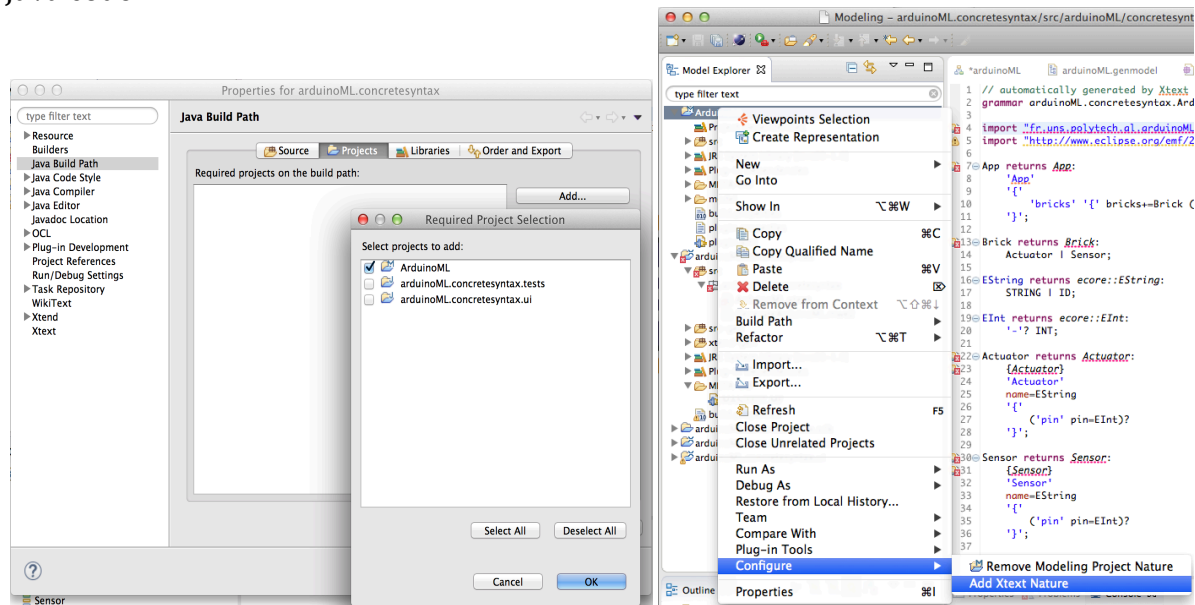


Create a new project using the 'Xtext Project from existing Ecore Models' option of the wizard.

Add the arduinoML.genmodel EPackage to the project and select 'App' as the entry rule (as it our root). Click 'Next' to setup the project name, the language name and the extension you choose for the model instances.



Once created, the project should automatically open the concrete ‘*.xtext’ syntax file. For it to know the concepts defined in your ecore file, you need to add the EMF project to the class path and add the Xtext nature to the initial project containing the generated Java code.

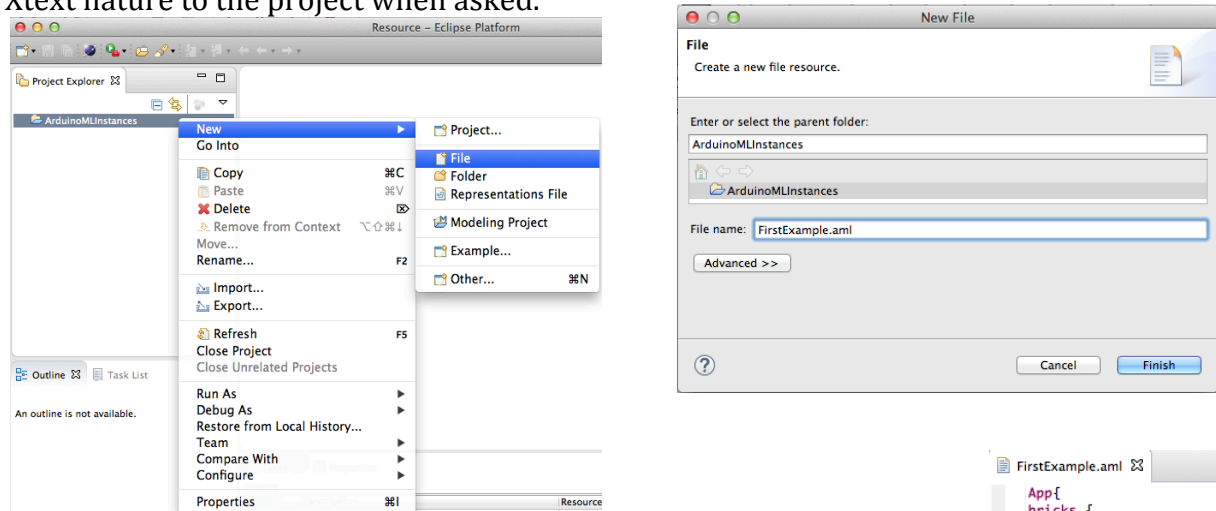


Next to the .xtext file, a .mwe2 file describe the workflow needed to build a runtime application using our language. Right-click on this file and run it as a MWE2 Workflow. If you are asked to use ANTLR3 parser, answer yes (y).

Once finished, the workflow should have generated several packages (formatting, generator, scoping, serializer and validation) in the Xtex project.

We are now able to launch a new runtime instance of Eclipse integrating our language as a plug in. Right-click on the Xtext project and select Run As > Eclipse Application.

A brand new Eclipse window is opened. Create a new Project ‘ArduinoMLInstances’ and then create a new File inside. It’s extension must be the one you decided before. Add the Xtext nature to the project when asked.



We can now instantiate ArduinoML model using a textual concrete syntax provided with auto-completion(Ctrl+Space) and pre-compiler features. Recreate our previous example.

```
App{
  bricks {
    Actuator red_led {
      pin 12
    },
    Sensor button {
      pin 8
    }
  }
}
```


Providing a new (better?) syntax

To edit the default syntax of our language, we have to work on the file which specify its grammar. Close the runtime eclipse and open the .xtext file.

We want to write things like “actuator red_led: 12”, so at the meta level :

```
"actuator" %NAME% ":" %PIN%
```

```

Actuator returns Actuator:
  {Actuator}
  'Actuator'
  name=EString
  '{'
    ('pin' pin=EInt)?
  '}'
→
Actuator returns Actuator:
  {Actuator}
  'Actuator' name=EString ':' pin=EInt;

```

Transform the actuator rule. Run the MWE2 workflow. Launch the runtime Eclipse.

We can now edit the App rule and factorize the previous change in the Brick rule.

```

1 // automatically generated by Xtext
2 grammar arduinoML.concretesyntax.ArduinoML with org.eclipse.xtext.common.Terminals
3
4 import "fr.uns.polytech.al.arduinoML"
5 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
6
7 App returns App:
8     'app'
9     '{'
10         'bricks'
11         bricks+=Brick
12         (bricks+=Brick)*
13     '}' ;
14
15 Brick returns Brick:
16     (Actuator | Sensor) name=EString ':' pin=EInt;
17
18 Actuator returns Actuator:
19     {Actuator}
20     'Actuator';
21
22 Sensor returns Sensor:
23     {Sensor}
24     'Sensor';
25
26 EString returns ecore::EString:
27     STRING ID;
28
29 EInt returns ecore::EInt:
30     '-'? INT;

```

Step #1.3: Generating code

Earlier we used the .genmodel file to generate java code. But what does EMF generate exactly? For each EClass are generated one Java Interface (in src/arduinoML) and one class implementing this interface (in src/arduinoML.impl). Note that all interfaces extend directly or transitively EObject (EMF equivalent of java.lang.Object). In src/arduinoML.util an ArduinoMLSwitch.java class has also been generated, we extend it to implement a Visitor Pattern in order to generate different pieces of code in for each meta classes.

We will also need to interpret a textual model to recover the proper defined domain hierarchy and then pass it to the visitor.

Visitor implementation

In a new package `arduinoML.homemade`, we create a new class `ArduinoMLSwitchPrinter` which extends the generated `Switch` and redefine the 'case*' methods to implement the desired behavior.

```
public class ArduinoMLSwitchPrinter extends ArduinoMLSwitch<String> {
```

Here, we want it to generate native Arduino code. To simplify we will print the code on the standard output. The 'doSwitch' method is responsible of the dispatch between those definitions, so it is this function we have to call, giving the root of our model as input (an `App`) to browse its containment tree.

```
    public String caseApp(App object) {
        StringBuilder sb = new StringBuilder();
        sb.append("// Code generated by ArduinoML\n"
            + "void setup() {\n");
        for(Brick b : object.getBricks())
            sb.append(doSwitch(b));
        sb.append("\n}");
        return sb.toString();
    }
```

An `App` instance generates a setup function containing its bricks. As the code generated by an `App` has to encapsulate the others concepts productions, the 'caseApp' method as to propagate the 'doSwitch' call on its children.

```
    public String caseActuator(Actuator object) {
        return "\tpinMode("+object.getPin()+", OUTPUT);\n";
    }
```

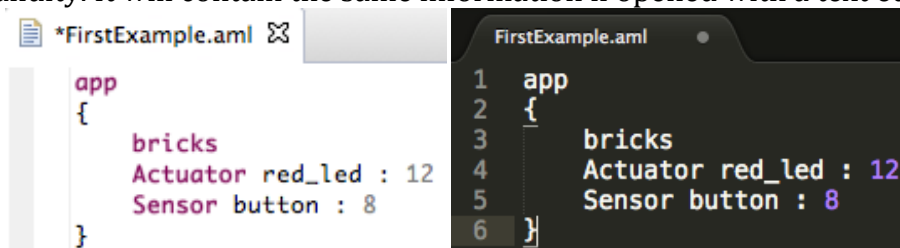
Each brick declare a `pinMode` and specify if it's an input or output one.

We now have a visitor pattern implemented to generate the relevant Arduino code, given the root of a valid ArduinoML model.

Note: if you copy paste the existing function definition from the `Switch` to your extending `Switch`, you should delete the '@generated' annotation.

Textual model parsing

When we create an instance model through the runtime Eclipse, the produce file is really only text, i.e. a dsl script, which is dynamically interpreted by our language plug-in to check its validity. It will contain the same information if opened with a text editor.



Once produce, we need to parse it to store our model in a format understandable by EMF. The standard format is `.xmi`.

The method '`ArduinoML2xmi(String modelPath, String destinationPath)`' is provided for this task in the Main class of the project '`ArduinoML2NativeArduino`'.

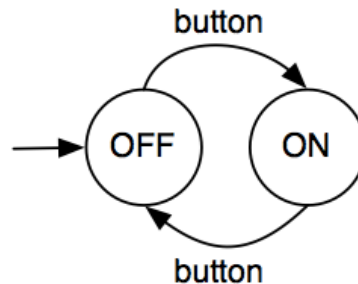
Exploit the .xmi format

The method 'xmi2NativeArduino(String xmiPath)' provides a way to load an xmi file and launch the visitor on its root to obtain the Arduino code.

```
Problems @ Javadoc Declaration Console
<terminated> Main [Java Application] /Library/Java/JavaVirtual
// Code generated by ArduinoML
void setup() {
    pinMode(12, OUTPUT);
    pinMode(8, INPUT);
}
```

Phase #2: Supporting behavioral concepts

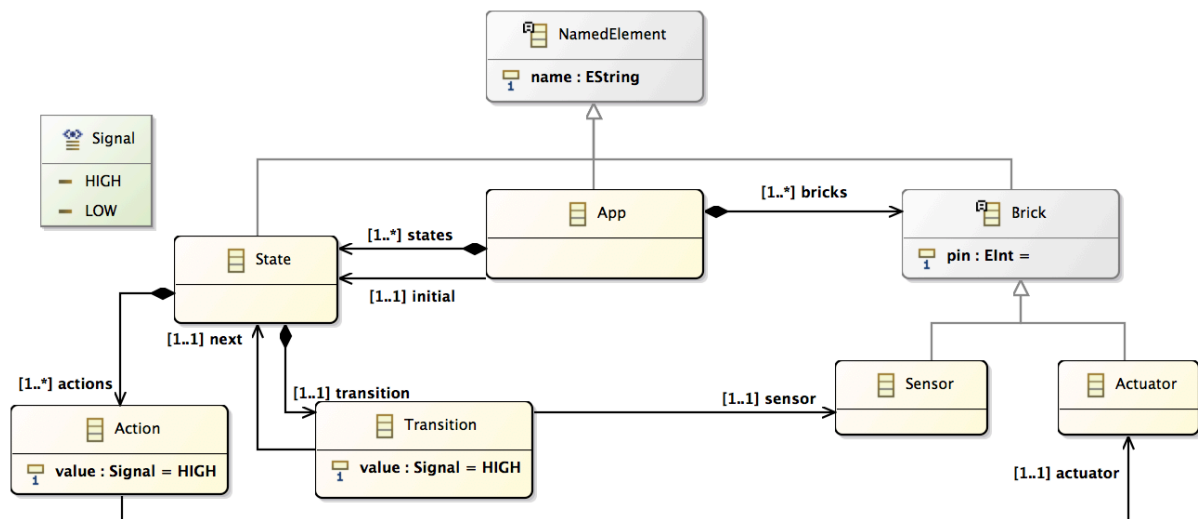
The behavior to be uploaded on the board can be modeled as an automaton.



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”.



Reload the 'genmodel' file (Right-click > Reload...). Open it and re-generate the model code (Right-click on ArduinoML node > Generate Model Code, cf. page 7).

At this point, our concrete syntax project should have some errors because we edited existing meta classes (Brick). It will be resolved when we will run the WE2 workflow to generate the proper assets for Xtext.

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

But first, we want to edit the concrete syntax in the ‘ArduinoML.xtext’ file to adapt our grammar and take into account our new concepts. We can create a temporary Xtext project based on our new ecore to use the default grammar as a basis and merge it into our existing one.

```
7@App returns App:
8  'app'
9  '{'
10    'bricks'
11    bricks+=Brick
12    (bricks+=Brick)*
13  '}'
14
15@Brick returns Brick:
16  (Actuator | Sensor) name=EString ':' pin=EInt;
17
18@Actuator returns Actuator:
19  {Actuator}
20  'Actuator';
21
22@Sensor returns Sensor:
23  {Sensor}
24  'Sensor';
25
26@EString returns ecore::EString:
27  STRING | ID;
28
29@EInt returns ecore::EInt:
30  '-'? INT;
31
32@/*App returns App:
33  'App'
34  name=EString
35  '{'
36    'initial' initial=[State|EString]
37    'bricks' '{' bricks+=Brick ( "," bricks+=Brick)* '}'
38    'states' '{' states+=State ( "," states+=State)* '}'
39  '}'
40
41 State returns State:
42  'State'
43  name=EString
44  '{'
45    'actions' '{' actions+=Action ( "," actions+=Action)* '}'
46    'transition' transition=Transition
47  '}'
48
49 Action returns Action:
50  'Action'
51  '{'
52    'value' value=Signal
53    'actuator' actuator=[Actuator|EString]
54  '}'
55
56 Transition returns Transition:
57  'Transition'
58  '{'
59    'value' value=Signal
60    'next' next=[State|EString]
61    'sensor' sensor=[Sensor|EString]
62  '}'
63
64 enum Signal returns Signal:
65  HIGH = 'HIGH' | LOW = 'LOW';*/

7@App returns App:
8  'app'
9  '{'
10    'bricks'
11    bricks+=Brick
12    (bricks+=Brick)*
13
14    'states' '{' states+=State ( "," states+=State)* '}'
15    'initial' initial=[State|EString]
16  '}'
17
18@Brick returns Brick:
19  (Actuator | Sensor) name=EString ':' pin=EInt;
20
21@Actuator returns Actuator:
22  {Actuator}
23  'Actuator';
24
25@Sensor returns Sensor:
26  {Sensor}
27  'Sensor';
28
29@EString returns ecore::EString:
30  STRING | ID;
31
32@EInt returns ecore::EInt:
33  '-'? INT;
34
35@State returns State:
36  'State'
37  name=EString
38  '{'
39    'actions' '{' actions+=Action ( "," actions+=Action)* '}'
40    'transition' transition=Transition
41  '}'
42
43
44@Action returns Action:
45  'Action'
46  '{'
47    'value' value=Signal
48    'actuator' actuator=[Actuator|EString]
49  '}'
50
51@Transition returns Transition:
52  'Transition'
53  '{'
54    'value' value=Signal
55    'next' next=[State|EString]
56    'sensor' sensor=[Sensor|EString]
57  '}'
58
59@enum Signal returns Signal:
60  HIGH = 'HIGH' | LOW = 'LOW';
```

Our new grammar is taking into account the structural modifications we performed in our meta model, so we can now regenerate the corresponding code to get rid of the errors in src-gen, to do so we run the mwe2 file.

Then, we launch the runtime Eclipse application to edit our previously defined model with the default rules. We need to add 2 states, and transitions from one state to the other.

```

FirstExample.aml
app RedButton
{
  bricks
  Actuator red_led : 12
  Sensor button : 8

  states {
    State off {
      actions {
        Action { value LOW actuator red_led }
      }
      transition Transition { value HIGH next on sensor button }
    },
    State on {
      actions {
        Action { value HIGH actuator red_led }
      }
      transition Transition { value HIGH next off sensor button }
    }
  }
  initial off
}

```

The default editor is definitively not suited for our needs!

Step #2.3: Providing a proper concrete syntax

We need to define better rules for Actions, Transitions, State and to edit the App one. For App, we may want to place the definition of the initial state right after the name, and to apply the same layout for states that we did for bricks.

```

7 App returns App:
8   'app' name=EString 'initial state' initial=[State|EString]
9   '{'
10    'bricks'
11    bricks+=Brick
12    (bricks+=Brick)*
13
14    'states'
15    states+=State
16    ( states+=State)*
17   '}'

```

Then, we want to be 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*

```

44 Action returns Action:
45   actuator=[Actuator|EString] '<=' value=Signal;
46
47 Transition returns Transition:
48   sensor=[Sensor|EString] 'is' value=Signal '>=' next=[State|EString];

```

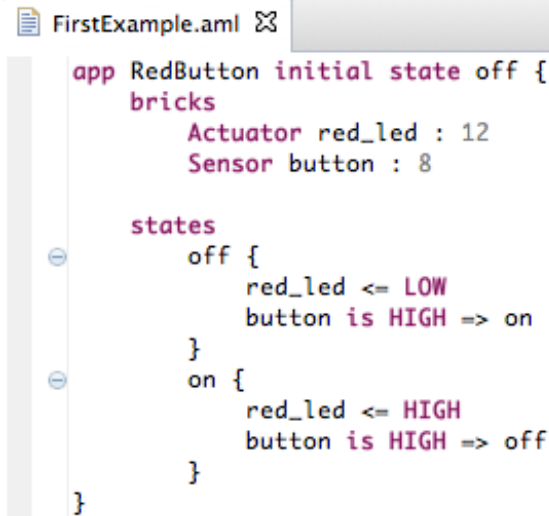
The State rule will basically define the name of the state, and provide a vertical list of actions, followed by its transition. We suppress the noise created by too much scoping by deleting the extra brackets.

```

36 State returns State:
37   name=EString '{'
38   actions+=Action
39   (actions+=Action)*
40   transition=Transition
41   '}'

```

With these rules defined and after regenerate the code assets through the workflow execution launch the runtime Eclipse and open the previously defined model to adapt it.

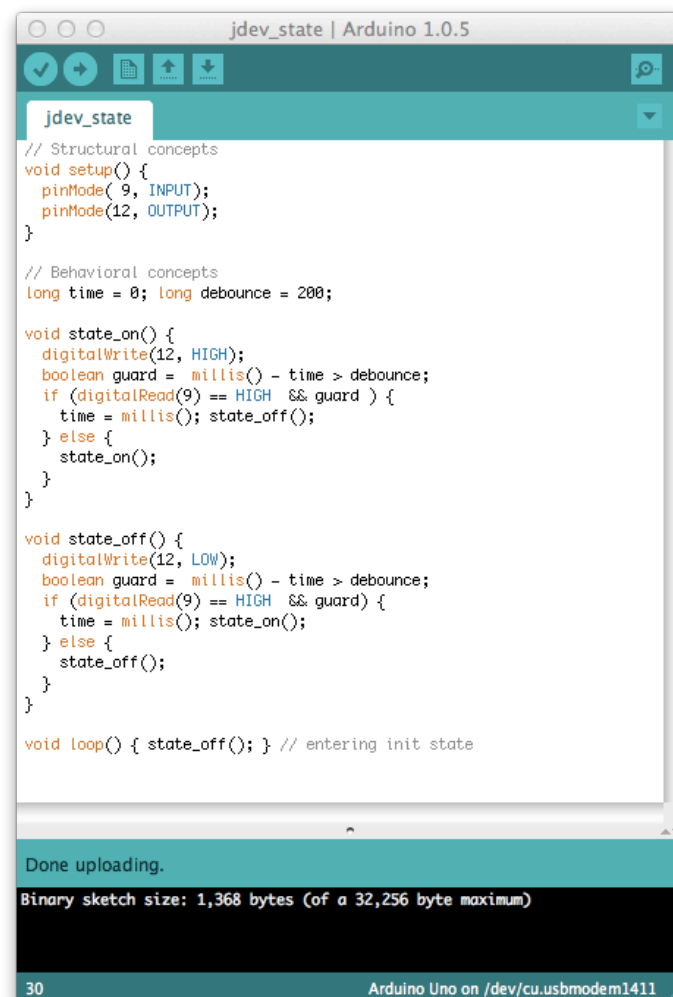


```
FirstExample.aml
app RedButton initial state off {
  bricks
    Actuator red_led : 12
    Sensor button : 8

  states
    off {
      red_led <= LOW
      button is HIGH => on
    }
    on {
      red_led <= HIGH
      button is HIGH => off
    }
  }
}
```

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 visitor case

First, we edit the App root case to add the needed propagation calls. 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.

```
public String caseApp(App object) {
    StringBuilder sb = new StringBuilder();
    sb.append("// Code generated by ArduinoML\n"
        + "// Structural concepts\n"
        + "void setup() {\n");
    for(Brick b : object.getBricks())
        sb.append(doSwitch(b));
    sb.append("}\n\n"
        + "//Behavioral concepts\n"
        + "long time=0; long debounce = 200;\n\n");
    for(State s : object.getStates())
        sb.append(doSwitch(s));
    sb.append("\nvoid loop() {state_"+object.getInitial().getName()+"();} // Entering init state");
    return sb.toString();
}
```

The “init_state” string in the “loop” function is easy to handle as we can go through our model to get the corresponding identifier.

Visit the Actions Case

We then edit the case method dedicated to the “Action” meta class. Its implementation is straightforward, retrieving the pin number of the actuator and the signal value to send.

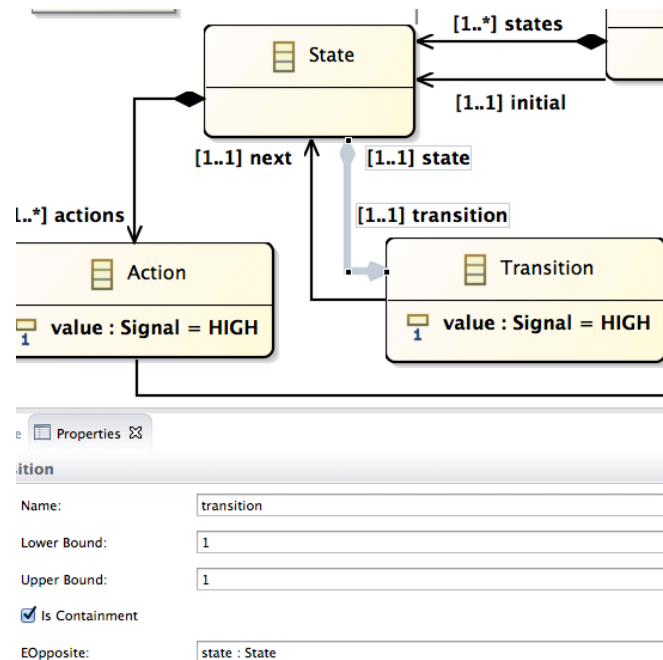
```
public String caseAction(Action object) {
    StringBuilder sb = new StringBuilder();
    sb.append("\tdigitalWrite("
        +object.getActuator().getPin()
        +", "+object.getValue().getLiteral()
        +");\n");
    return sb.toString();
}
```

Visit the Transitions Case

The template is a little longer due to the guard, but quite straightforward to write too, using similar accessing methods.

```
public String caseTransition(Transition object) {
    StringBuilder sb = new StringBuilder();
    sb.append("\tif (digitalRead(" + object.getSensor().getPin() + ") == " + object.getValue().getLiteral() + " && guard ) {"
        + "\t\ttime = millis(); state_"+object.getNext().getName()+"();\n"
        + "\t} else { state_"+object.getState()+"(); }\n");
    return sb.toString();
}
```

The main issue here is the “getState()” to be used in the “else” condition. This state is currently unknown at the transition level. We need to edit our model to make the relation between a state and its transition bi-directional. To do so, add a reference from Transition to State, named state, and specify it as the opposite of the containment relation ‘transition’ from State to Transition.



Reload the 'genmodel' file, regenerate the model code from it. The error in our 'caseTransition' method should be resolved, so you access its name.

```
public String caseTransition(Transition object) {
    StringBuilder sb = new StringBuilder();
    sb.append("\tif (digitalRead(" + object.getSensor().getPin() + ") == " + object.getValue().getLiteral() + " && guard ) {\n"
        + "\t\ttime = millis(); state_" + object.getNext().getName() + "();\n"
        + "\t} else { state_" + object.getState().getName() + "(); }\n");
    return sb.toString();
}
```

Visit the State Case

Nothing new here, accessing the attribute values and propagating the doSwitch on the children does the trick.

```
public String caseState(State object) {
    StringBuilder sb = new StringBuilder();
    sb.append("void state_" + object.getName() + "()\n");
    for(Action a : object.getActions())
        sb.append(doSwitch(a));
    sb.append("\tboolean guard = millis() - time > debounce;\n"
        + "\tdoSwitch(object.getTransition())\n"
        + "\t}\n");
    return sb.toString();
}
```

Generating ArduinoML code

Run the previously modeled application into the main of the ArduinoML2NativeArduino project and you should get the following result. That's all folks!

Problems Console Properties

```
<terminated> Main [Java Application] /Library/Java/JavaVirtualMach  
// Code generated by ArduinoML  
// Structural concepts  
void setup() {  
    pinMode(12, OUTPUT);  
    pinMode(8, INPUT);  
}  
  
//Behavioral concepts  
long time=0; long debounce = 200;  
  
void state_off() {  
    digitalWrite(12, LOW);  
    boolean guard = millis() - time > debounce;  
    if (digitalRead(8) == HIGH && guard ) {  
        time = millis(); state_on();  
    } else { state_off(); }  
}  
void state_on() {  
    digitalWrite(12, HIGH);  
    boolean guard = millis() - time > debounce;  
    if (digitalRead(8) == HIGH && guard ) {  
        time = millis(); state_off();  
    } else { state_on(); }  
}  
  
void loop() {state_off();} // Entering init state
```