03/02/2021

SI5 Domain Specific Languages
Report Project 1
External and internal DSL ArduinoML

Our team

Lydia BARAUKOVA (SI5 AL)
Jean-Marie DORMOY (SI5 WEB-IA)
Gabriel REVELLI (SI5 IHM)

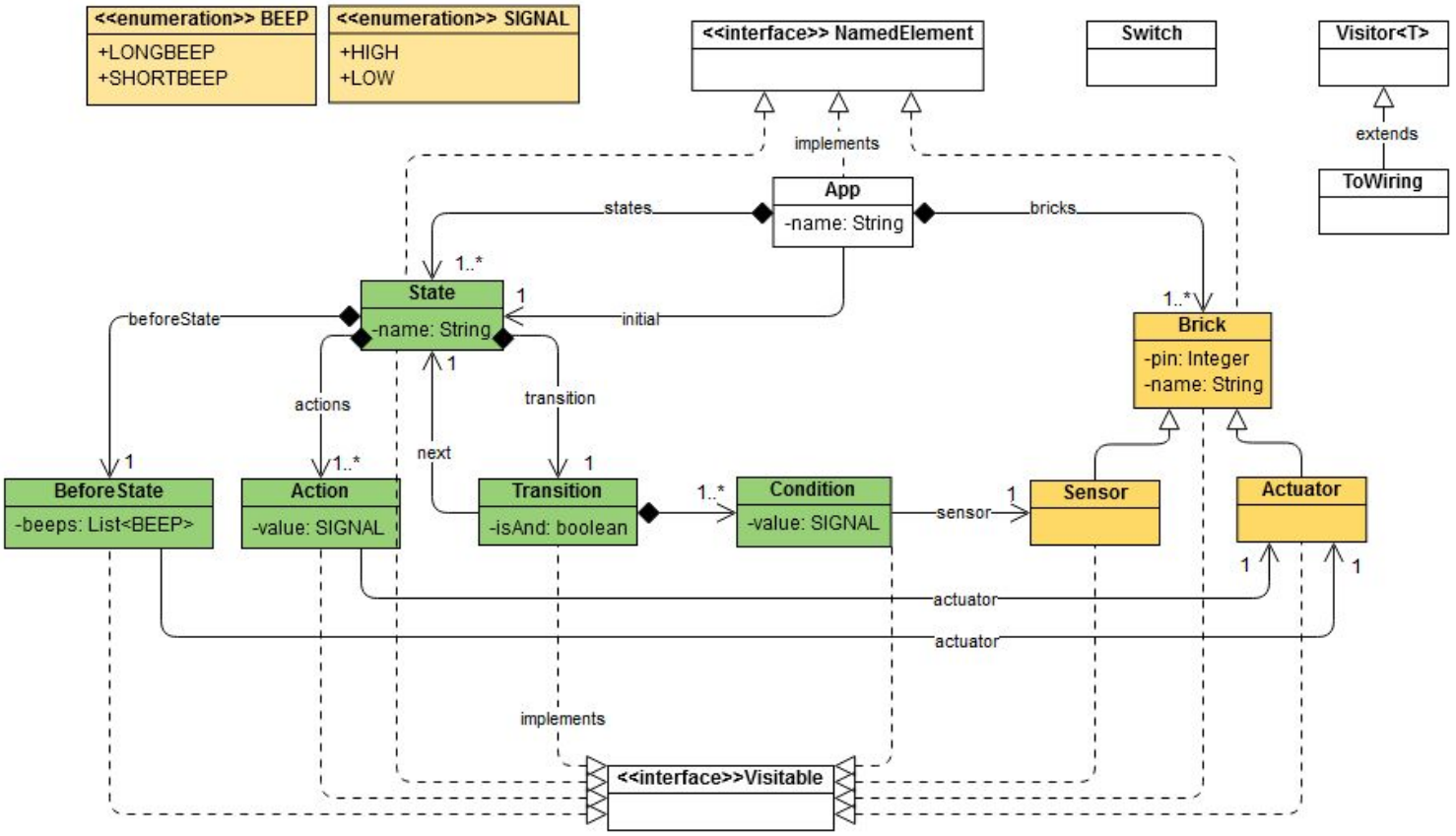
Link to our code

https://github.com/Livelinndy/si5-dsl-team-a/tree/main/Project1-ArduinoML

Description of the developed languages

Domain model

The domain model is the same for both external and internal DSL. You can find it in the folder named “kernel-jvm”. Here’s the class diagram of our domain model:



Concrete syntax (represented in a BNF-like form)

Internal DSL

```
<root>            ::= <bricks> <states> <before> <initial> <transitions> <export> <EOF>

<bricks>          ::= ( <sensor> | <actuator> )+
<sensor>          ::= "sensor" <location>
<actuator>        ::= "actuator" <location>
<location>        ::= <IDENTIFIER> "pin" <PORT_NUMBER>

<states>          ::= <state>+
<state>           ::= "state" <IDENTIFIER> "means" <action>
<action>          ::= <IDENTIFIER> "becomes" <SIGNAL> <recursiveAction>*
<recursiveAction> ::= "and" <IDENTIFIER> "becomes" <SIGNAL>

<before>          ::= "before" <IDENTIFIER> "with" <IDENTIFIER> "make" <beep>
<beep>            ::= ( ""shortbeep"" | ""longbeep"" )

<initial>         ::= "initial" <IDENTIFIER>

<transitions>     ::= <transition>+
<transition>      ::= "from" <IDENTIFIER> "to" <IDENTIFIER> "when" <condition>
<condition>       ::= <IDENTIFIER> "becomes" <SIGNAL>

<export>          ::= "export" <IDENTIFIER>

<PORT_NUMBER>     ::= <DIGIT>[1-2]?
<IDENTIFIER>      ::= """ <LOWERCASE> ( <LOWERCASE> | <LOWERCASE> | <DIGIT> )+ """
<SIGNAL>          ::= ""high"" | ""low""
<LOWERCASE>       ::= [a-z]
<UPPERCASE>       ::= [A-Z]
<DIGIT>           ::= [0-9]

<NEWLINE>         ::= ("\r"? "\n" | "\r")+      -> skip
<WHITESPACE>      ::= ((" " | "\t")+)           -> skip
<COMMENT>         ::= "#" ~( "\r" | "\n" )*     -> skip
```

External DSL

```
<root>        ::=       <declaration> <bricks> <states> <EOF>

<declaration> ::= "application" <IDENTIFIER>

<bricks>      ::= ( <sensor> | <actuator> )+
<sensor>      ::= "sensor" <location>
<actuator>    ::= "actuator" <location>
<location>    ::= <IDENTIFIER> ":" <NUMBER>

<states>      ::= <state>+
<state>       ::= <initial>? <IDENTIFIER> "{"  <beep>? <action>+ <transition> "}"
<beep>        ::= <IDENTIFIER> <BEEP_TYPE> <NUMBER>
<action>      ::= <IDENTIFIER> "<=" <SIGNAL>
<transition>  ::= <IDENTIFIER> "is" <SIGNAL> <condition>* "=>" <IDENTIFIER>
<condition>   ::= <BOOL> <IDENTIFIER> "is" <SIGNAL>

<initial>     ::= "->"

<NUMBER>      ::= <DIGIT> | [1][0-3]
<IDENTIFIER>  ::= <LOWERCASE> ( <LOWERCASE> | <LOWERCASE> | <DIGIT> )+
<SIGNAL>      ::= "HIGH" | "LOW"
<BOOL>        ::= "&"    | "|"
<BEEP_TYPE>   ::= "LONGBEEP" | "SHORTBEEP"

<LOWERCASE>   ::= [a-z]
<UPPERCASE>   ::= [A-Z]
<DIGIT>       ::= [0-9]
<NEWLINE>     ::= ("\r"? "\n" | "\r")+      -> skip
<WHITESPACE>  ::= ((" " | "\t")+)           -> skip
<COMMENT>     ::= '#' ~( '\r' | '\n' )*     -> skip
```

Extension

Description of our extension

Signaling stuff by using sounds. The goal of this extension is to allow the user to define a state machine in which some actions make use of the buzzer. Such actions may for instance define two short beeps or one long, as defined by the user. This is a kind of behavior used to signal the default of a motherboard in a PC.

Acceptance scenario: In order to request attention from his users, Donald wants to definethat an entry in a state emits three consecutive short beeps. Also, when entering in a state meaning that the process is finished, Donald wants to emit a long beep.

How it was implemented

Modifications in the kernel

- Created BEEP enum with values LONGBEEP, SHORTBEEP. It corresponds to a row of signals that can be played by an actuator.
- Created class *BeforeState* containing a list of BEEP and an *Actuator* and added the corresponding *visit* abstract method's signature in *Visitor* class.
- Implemented *visit* method taking a *BeforeState* object in argument in class *ToWiring.*

Modifications in the internal DSL

We added a new phase just before a state and called it BeforeState (which is an object). This object has in attribute a list of BEEP and an actuator to give the signal to. In internal DSL, we've added 4 new terms like "before", "with", "make" and "and". This terms are used like this :

```
before "off" with "buzzer" make "longbeep" and "shortbeep" and "longbeep"
```

As you can see "before" have, as a parameter, a State, "with" an Actuator and "make" and "and" have a BEEP as a parameter. We also had to change the GroovuinoMLBasescript groovy class to recognize the new syntax we've added.

```
def before(state) {
    [with: { actuator ->
        List<BEEP> beeps = new ArrayList<BEEP>()
        ((GroovuinoMLBinding) this.getBinding()).getGroovuinoMLModel().createBeforeState(
            state instanceof String ? (State)((GroovuinoMLBinding)this.getBinding()).getVariable(state) : (State)state,
            actuator instanceof String ? (Actuator)((GroovuinoMLBinding)this.getBinding()).getVariable(actuator) : (Actuator)actuator,
            beeps)
        def close
        close = { beep ->
            beeps.add(beep instanceof String ? (BEEP)((GroovuinoMLBinding)this.getBinding()).getVariable(beep) : (BEEP)beep)
            [and: close]
        }
        [make: close]
    }]
}
```

The declaration is recursive because as you can see, we can add as many sort of beeps we want that are stored in the BeforeState's BEEP list.
We can personnalise the sound we want to have before a given state.

Modifications in the external DSL

```
states          :   state+;
    state       :   initial? name=IDENTIFIER '{'  beep* action+ transition '}';
    beep        :   actuatorId=IDENTIFIER type=BEEP_TYPE quantity=DIGIT_NUMBER?;
    action      :   receiver=IDENTIFIER '<=' value=SIGNAL;
    transition  :   trigger=IDENTIFIER 'is' value=SIGNAL condition* '=>' next=IDENTIFIER ;
    condition   :   booleanOperator=BOOL trigger=IDENTIFIER 'is' value=SIGNAL ;
    initial     :   '->';
```

```
on {
    buzzer LONGBEEP
    buzzer SHORTBEEP 3
    led <= LOW
    buzzer <= HIGH
    button is HIGH => off
}
```

We adjusted the external DSL grammar by adding a *beep* rule, to enable introduction of instructions like “buzzer LONGBEEP 2” at the beginning of a *state*.

Modifications in class *ModelBuilder*:

When we enter into a *state*, we create the corresponding object which we store in *currentState* and we create a new object *BeforeState* that will be stored in *currentBeforeState*.

Inside the *enterBeep* method, we use its *ArduinomlParser.BeepContext* argument in order to retrieve the *BEEP* type and *x*, the number of *BEEPs*, then we add *x* times the corresponding type to the list of *BEEP* of *currentBeforeState*.

When we exit a *state*, *currentBeforeState* is stored in the *beforeState* field of *currentState*. The field *currentState.beforeState* now contains a *BeforeState* object itself containing the list of *BEEPs* to produce. To finish, we add *currentState* to the list of *states* of the application.

At the end of the methods’ execution of class *ModelBuilder*, we have generated all the variables corresponding to the *.arduinoml* program given in input. The only thing left to do is to “visit” these variables inside class *ToWiring*.

Implemented 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 buttons stops the sound.
3. **State-based alarm**. Pushing the button once switches 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 stops the buzzer and switches the LED on. Pushing it again switches the LED off, and makes the system ready to make noise again after one push, and so on.
5. **Signaling stuff by using sounds**. The goal of this extension is to allow the user to define a state machine in which some actions make use of the buzzer.

Critical analysis

1) Of the DSL implementation with respect to the ArduinoML use case:

During the implementation of scenario 2, we have added only 2 logical operators: “and” and “or”. Also, in the current version we can’t mix logical operators. Such implementation was enough to cover our scenarios, but we are aware that the DSL is not complete without other logical operators (such as “not”, for example) and the possibility to mix them.

For scenario 5, we can only put buzzer sounds at the beginning of a state. Also, for LONGBEEP and SHORTBEEP we must specify the actuator (buzzer) pin. But if the user doesn’t actually connect a buzzer to this pin, there will be no sound.

2) Of the technology we chose to achieve it:

Antlr is a proven Java Library which allowed us to modify a reduced number of files and to gain efficiency while implementing the differents parts of the project:

- the file .g4 and its corresponding grammar.
- the *ModelBuilder* class which analyzes tokens coming from the lexical analysis (thanks to the context object given in argument to *ModelBuilder*’s methods which visit the tokens) and that builds all the data structures containing information extracted from tokens. These data structures are then passed to the kernel and to the class *ToWiring* which is responsible for generating the *.ino* code.

Groovy is a Java based object oriented programming language that allows us to recognise syntax with less effort than in java. We can do complex things like parsing a text in an easy way, with a map (we used this system into the BaseScript class). The language supports closures (lambdas) and is interoperable with java, it means that we can set some files with a “.groovy” extension and others with “.java” and it’ll still compile. As an example, we Bind values and types in the Binding class (java), we parse the scripts with maps in the BaseScript class (groovy) construct models (java) Groovy is more lenient than Java.

The 2 above mentioned technologies are based on Java, so it seemed intuitive for us to use a kernel written in Java as well. That way we didn't have to have 2 different kernels. Also, writing all the project parts in the same language allowed us to focus more on the concepts needed to implement coherent DSLs instead of focusing on the programming language.

Responsibilities of each team member

Lydia BARAUKOVA	External DSL, kernel, report, initial setup
Jean-Marie DORMOY	External DSL, kernel, report
Gabriel REVELLI	Internal DSL, kernel, report

Tinkercad setup

Here's our setup for testing the scenarios. We have connected pins 4 and 7 to the same button to be able to test scenario 2. In real life conditions we would have used 2 different buttons.
https://www.tinkercad.com/things/lWooHwaZVN7-grand-densor/editel?sharecode=ZztOljhf3Ef5CwISBvGqP_jznG9YpFPGcZQY-47D4Ow