Université Côte d'Azur

ArduinoML

—

Domain Specific Language

Github : `https://github.com/FontanyLegall-Brandon/DSL`

Auteurs :
Michel Marmone–Marini
Aymeric Valdenaire
Brandon Fontany–Legall
Thomas Mahe

2 février 2020

Table des matières

1 Présentation du projet **1**
1.1 Sujet . . . 1
1.2 Code source et exécution . . . 1
1.3 Equipe . . . 1
1.4 Scénarios . . . 1
1.4.1 Scénarios simple . . . 1
1.4.2 Scénarios complexe . . . 2

2 Conception **2**
2.1 Choix technologique . . . 2
2.1.1 Choix technologique pour le DSL interne . . . 2
2.1.2 Choix technologique pour le DSL externe . . . 2
2.2 Conception . . . 3
2.2.1 Structure Kernel . . . 3
2.2.2 DSL interne . . . 3
2.2.3 DSL externe . . . 5

3 Ouverture **7**

Chapitre 1

Présentation du projet

1.1 Sujet

Ce projet consiste en la réalisation d'un DSL (Domain Specific Language) interne ainsi qu'un DSL externe pour permettre une écriture simplifiée de code Arduino.

1.2 Code source et exécution

Le code source de nos DSL est présent au lien suivant : https://github.com/FontanyLegall-Brandon/DSL

Pour l'exécution :

- Java : Exécuter la class Main.java. Cette dernière créera les fichiers .ino dans le dossier out avec les différents ArduinoApp défini en interne dans cette même classe
- Python : Exécuter main.py créera les fichiers .ino dans le dossier out avec les différents samples .aml

1.3 Equipe

Notre équipe est composé de

- Thomas Mahe : DSL Externe
- Michel Marmone–Marini : DSL Externe
- Aymeric Valdenaire : DSL Interne
- Brandon Fontany–Legall : DSL Interne

1.4 Scénarios

1.4.1 Scénarios simple

Nous avons ainsi implémenter notre propre language ainsi que certains scénarios simple pour prouver que notre code Arduino est fonctionnel.

- Création d'une alarme simple : Lorsque nous appuyons sur un bouton, ce dernier enclenche une LED ainsi qu'un buzzer. Relâcher le bouton éteint le beau/buzzer.
- Création d'une alarme à double vérification : Un signal sonore est déclenché si et seulement si les deux boutons sont enfoncés en même temps. Relâcher au moins un des boutons arrête le son.
- Création d'une alarme basée sur un état : La pression sur un bouton allume le système, une nouvelle pression l'éteint.
- Alarme multi-états : La pression sur un bouton déclenche le buzzer. Appuyer à nouveau sur le bouton arrête le butter et allume la led. Appuyer encore une fois éteint la led et rend le système prêt à de nouveau déclencher le buzzer, et ainsi de suite.

1.4.2 Scénarios complexe

Nous avons aussi mis en place certains scénarios plus complexe.

— Spécifier la fréquence d'exécution (Specifying Execution Frequency)

— Communication à distance (Remote Communication)

— Affichage d'exception (Exception Throwing)

— Support pour l'écran LCD (Supporting the LCD screen)

— Prise en charge des briques analogiques (Handling Analogical Bricks)

Les définitions des scénarios acceptable par notre DSL externe sont visible ici : https://github.com/FontanyLegall-Brandon/DSL/tree/master/external/samples

Chapitre 2

Conception

2.1 Choix technologique

2.1.1 Choix technologique pour le DSL interne

Dans un premier temps, nous avons fait le tours des divers possibilibités de DSL interne. Nous avons notamment remarqué

— Groovy

— Java

— MPS

Nous avons choisit **Java** car nous avons pas mal d'expérience dans ce language. Nous avions pu choisir Groovy qui est assez proche mais, au vu du pattern Builder, qui est plus simple que l'écriture d'un nouveau Groovy Shell, nous nous sommes orienté vers Java.

2.1.2 Choix technologique pour le DSL externe

Pour l'implémentation de DSL externe, nous avons vu plusieurs possibilités comme

— MPS

— xtext

— textx

— lex/yacc

Nous avons choisi **Textx**. Textx est un meta-langage inspiré de Xtext permettant de créer des DSL avec Python. Il est notamment beaucoup plus simple que Xtext. Ce qui nous a le plus orienté sur cette décision est la possibilité d'avoir, concrétement, le code entre les mains et non, comme MPS ou Xtext, des fichiers pré-fait avec possiblement des limitations où au final, nous ne comprennons pas vraiment ce que nous ferions.

TextX suit la syntaxe et la sémantique de Xtext mais diffère à certains endroits.Il utilise l'analyseur Arpeggio PEG qui permet de ne pas avoir d'ambiguïtés grammaticales et d'avoir un regard sur ce que nous faisons.

2.2 Conception

2.2.1 Structure Kernel

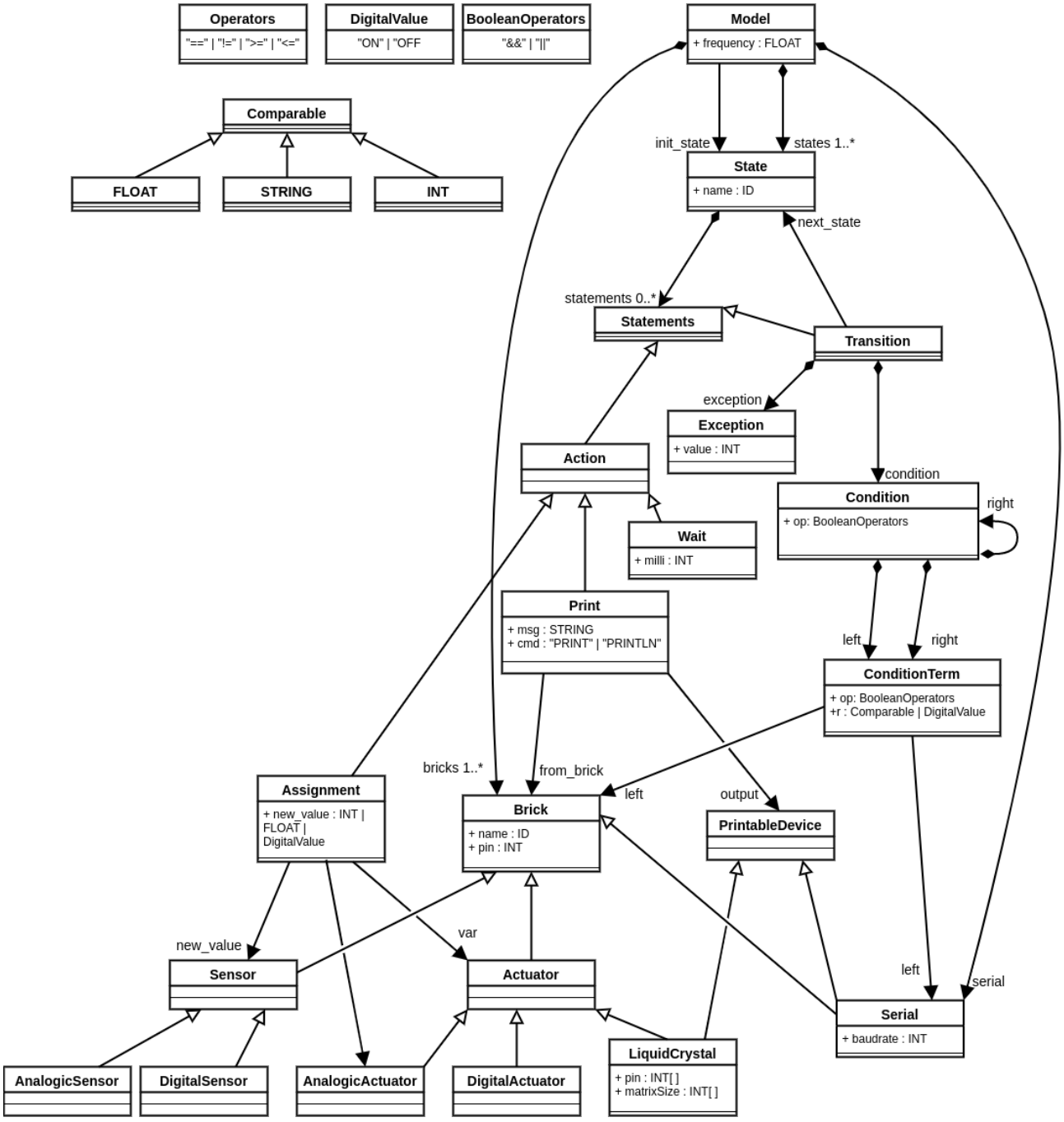

FIGURE 2.1 – Modèle de Domaine

Visible en pdf : `https://github.com/FontanyLegall-Brandon/DSL/raw/master/Kernel.pdf`

2.2.2 DSL interne

Pour la partie création du DSL, nous avons utilisé le pattern Builder. Le pattern Builder est un design pattern qui permet de construire des objets complex étape par étape.

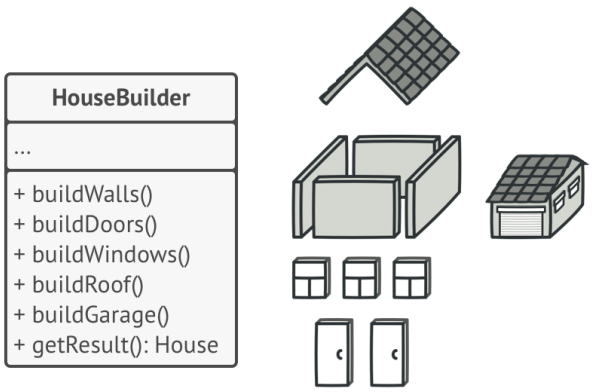

FIGURE 2.2 – Builder pattern (https ://refactoring.guru/design-patterns/builder)

Il nous permet, entre autres, de créer notre objet ArduinoApp à l'aide d'appels successifs. De plus, ce pattern nous permet de faire un language strict empêchant le développeur à utiliser des méthodes dans des cas interdit.

Voici, en exemple, le code du premier scénario simple avec le pattern builder :

```
ArduinoApp arduinoApp =
        arduino("scenario1","off")
            .setup(digitalSensor("button", 10))
            .setup(digitalActuator("led", 11))
            .setup(digitalActuator("buzzer", 9))
              .states()
                .state("off")
                  .set("led").toLow()
                  .set("buzzer").toLow()
                  .when().ifIsEqual("button", "ON").thenGoToState("on")
                .state("on")
                  .set("led").toHigh()
                  .set("buzzer").toHigh()
                  .when().ifIsEqual("button", "OFF").thenGoToState("off")
          .build();

Generator generator = new Generator();
arduinoApp.accept(generator);
System.out.println(generator.getGeneratedCode());
```

2.2.3 DSL externe

Description EBNF

La description EBNF est disponible ici : http://hamlab.fr/dsl/

Il est possible de cliquer sur les éléments inclus dans une description pour être redirigé vers sa définition.

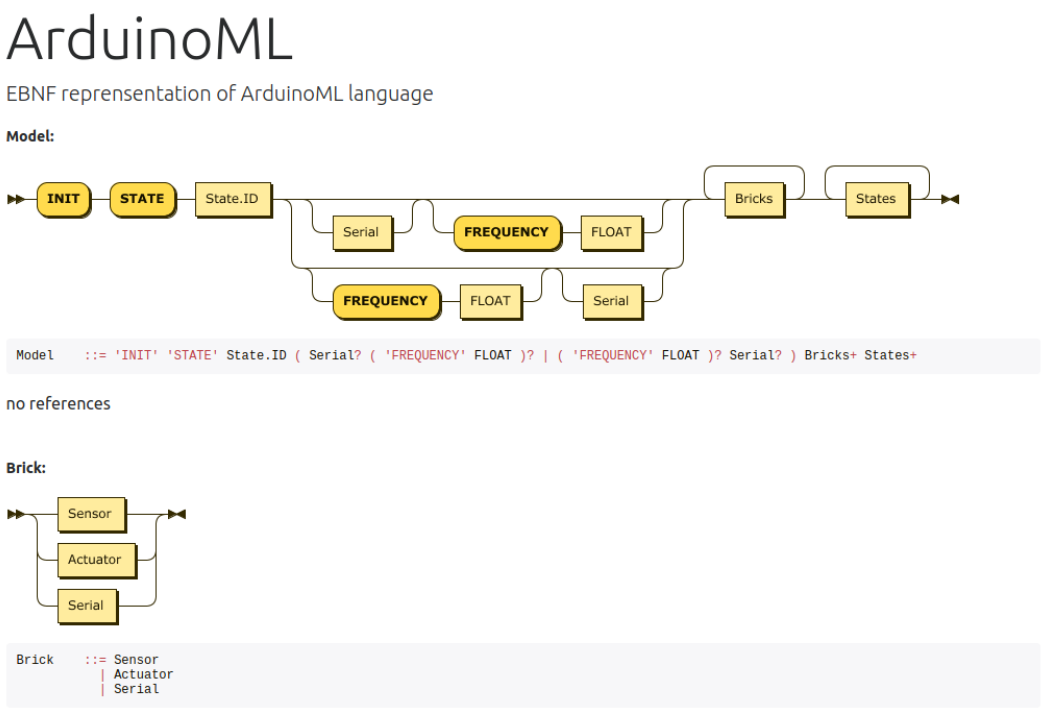

FIGURE 2.3 – Preview

Implémentation

Notre DSL externe consiste en la réalisation d'un metamodel généré à partir d'une grammaire et d'un ensemble de classe Python faisant lien avec les symboles de la grammaire. Une fois ce métamodel généré, un fichier peut être consommé par le métamodel afin d'en produire ou non du code.

Nous utilisons la méthode de Traduction dirigé par la syntaxe pour générer le code arduino, ainsi pour un ensemble de symboles de notre grammaire, l'objet qui lui est associé doit implémenter la méthode $\_\_str\_\_$.

textX propose une mécanique très pratique qui permet au sein de la grammaire de définir des références vers d'autres objects. Exemple pour une grammaire de la forme :

```
// Complex number rule
Complex :
    'var' name=ID '=' real=FLOAT 'i' imaginary=FLOAT |
    'var' name=ID '=' Addition
;
Addition :
    l=[Complex] '+' r=[Complex]
;
```

Ce langage acceptera l'entrée :

```
var a = 2.5i6.6
var b = 6.5i1.6

var c = a + b
```

En effet, le symbole `[Complex]` signale qu'il s'agit vers une référence vers un objet Complex, textX cherchera alors dans le modele, un objet de type Complex dont sont `ID` est égal à la référence. Textx lancera une erreur si il n'arrive pas à résoudre cette référence.

Dans l'exemple ci-dessus, l'objet associé à la règle `Addition` contiendra les champs `l` et `r` ces deux objets ce seront pas de type `ID` mais contiendront la valeur de la référence. Cette fonctionnalité prend également en compte l'héritage, permettant ainsi la mise en place d'un système de typage très facilement.

Nous avons fait usage de cette fonctionnalité notement pour pouvoir référencer des Brick dans les affectations ou les expressions booleennes.

Exemple d'un programme écrit dans notre langage :

```
INIT STATE off

DIGITAL SENSOR button 10
DIGITAL ACTUATOR led 11
DIGITAL ACTUATOR buzzer 9

STATE off {
    led = OFF
    buzzer = OFF
    button == ON => STATE on
}

STATE on {
    led = ON
    buzzer = ON
    button == OFF => STATE off
}
```

Ici la ligne `INIT STATE off`, fait référence à un état défini ci-dessous. A la génération du modele, si aucune instance State n'existe sous l'ID `"off"` le programme n'est pas validé.

Exemple de la définition d'une Brick

```
class Brick(object):
    def __init__(self, parent, name, pin):
        self.parent = parent
        self.pin = pin
        self.name = name

    def generate_var_init_code(self):
        return 'int {} = {};'.format(self.name, self.pin)

    def dependencies(self):
        return ""

class Sensor(Brick):
    def __init__(self, parent, name, pin):
        super().__init__(parent, name, pin)

    def generate_setup_code(self):
        return 'pinMode({}, INPUT);'.format(self.name, self.pin)

class DigitalSensor(Sensor):
    def __init__(self, parent, name, pin):
        super().__init__(parent, name, pin)

    def generate_setup_code(self):
        return super().generate_setup_code()

    def inline_read_code(self):
        return 'digitalRead({})'.format(self.name)
```

```
29
30      def __str__(self):
31          return '{}'.format(self.name)
```

On peut voir ici l'ensemble des méthodes communes aux Bricks ainsi que leurs champs, avec l'implémentation au sein des classes filles les méthodes spécifiques à l'instance. Plusieurs méthodes de représentation d'une Brick sont implémentées au sein des classes filles, à utiliser selon le context. Par exemple à l'intérieur d'une condition, il est nécessaire d'utiliser la méthode `inline_read_code` pour les instances Sensor contenues dans l'expression représentant la Condition.

Nous disposons d'un système de typage très sommaire pour les expressions de type Affectation, ainsi il n'est possible d'affecter un FLOAT qu'à une instance de type AnalogicActuator. Les valeurs de type INT ou bien obtenue depuis un Sensor peuvent cependant être assignés à un objet plus générique de type Sensor.

Une approche similaire est utilisé dans l'implémentation des expressions booleennes, seul un objet de type Serial peut être comparé à une String tandis qu'une Brick peut être comparé à un ensemble plus large de valeurs (Brick, INT, FLOAT).

Chapitre 3

Ouverture

Pour conclure, nous avons décider de faire nos DSL de manière à pouvoir écrire facilement des state machine ce qui se prête bien à la programmation Arduino.

Au niveau de nos choix technologiques, nous sommes globalement satisfaits des choix que nous avons faits. TextX, malgré le fait que la meta-langage soit en python avec donc un manque de typage, ce qui rend le code complexe à comprendre, il est cependant très simple de prise en main. Côté Java, la syntaxe de notre DSL en méthode chaining est beaucoup moins usé friendly mais cela nous permet de contrôler au maximum la syntaxe limitant l'utilisateur en matière d'utilisation de méthode dans des cas interdits.