DSL KICKLAB : ARDUINOML

Génovèse Matthieu Liechtensteger Michael Chennouf Mohamed

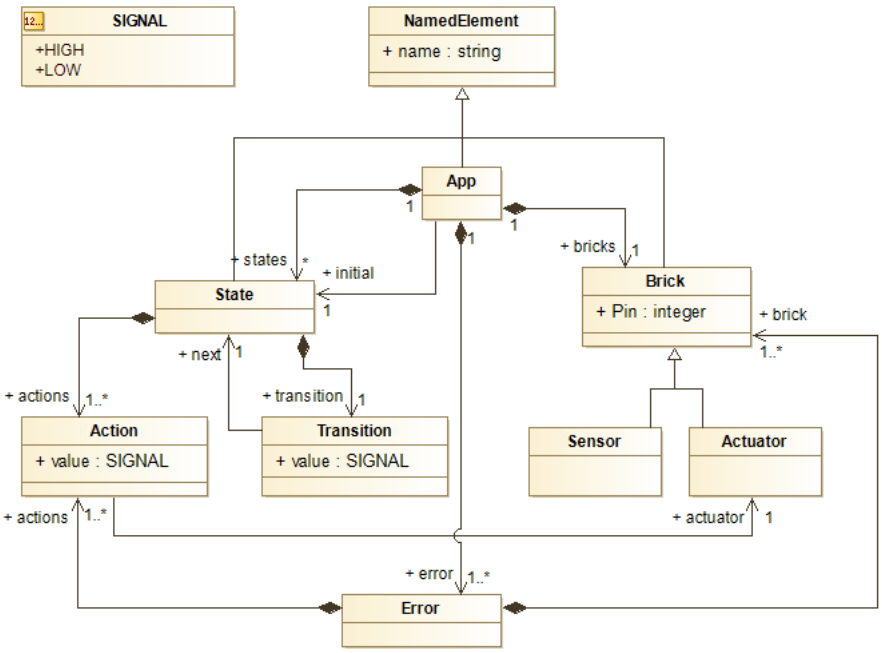
January 2018

1 Introduction

Ce projet consiste à réaliser un DSL (Domain Specific Langage) permettant à un utilisateur de générer du code Arduino.
Nous avions le choix entre plusieurs technologies afin de développer un DSL embarqué (interne à un langage de programmation large) et un DSL externe (liberté d'utiliser la forme que l'on souhaite). Nous avons choisi les langages Groovy pour notre DSL interne, et ANTLR pour notre DSL externe. Nous sommes partis des langages présents dans ArduinoML et les avons modifiés pour qu'ils puissent prendre en charge les scénarios de base demandés, plus une feature additionnelle qui consistait à pouvoir lever des exceptions dans le code si une certaine condition était rencontrée.

2 Description du langage

Voici le diagramme des composants de notre langage :



2.1 Déclarer les inputs / outputs

Dans notre langage nous commençons par déclarer les inputs / outputs utilisés par arduino :

```
sensor "name" pin "integer"
```

Voici comment on déclare un input en Groovy, il aura comme nom "name" et sera branché sur le pin indiqué.

```
actuator "name" pin "integer"
```

En Antlr, la déclaration de sensor et d'actuator se fait de façon similaire :

```
sensor "button" pin "integer"
```

Même principe que pour les inputs.

En Antlr, les inputs/ouputs sont générés de façon similaire, avec pour syntaxe :

```
sensor "name" : "integer"
```

Dans la grammaire, les inputs et outputs sont représentés par des objets appelés 'bricks', qui sont composés du mot clé 'sensor' ou 'actuator', puis du nom et d'un entier, pour relier l'objet à un nom et un pin sur l'arduino.

2.2 Déclarer les états

Ici on va déclarer les etats possibles pour notre système et leurs actions :

```
state "on" means "led" becomes "high" and "buzzer" becomes "high"
```

On définit un état "on" qui s'active lorsque nos 2 actuators reçoivent le signal "high" (signal est un objet de notre langage non définissable par l'utilisateur, il peut prendre les valeurs "high" et "low", cela fonctionne un peu comme les booléens). On aura ici au préalable déclaré 2 actuators ("led" et "buzzer"). Dans cette exemple, la partie 'and "buzzer" becomes "high"' est optionnelle, on a la possibilité d'ajouter plusieurs fois le mot end dans un seul état. Pour Antlr, la modification d'un actuator ou d'un sensor se fait via le symbole '<='.

2.3 Déclarer les transitions d'états

Lorsque tous les états de notre système sont bien définis, il faut déclarer les transistions qui permettent de passer de l'un à l'autre.

```
from "etat1" to "etat2" when "button" becomes "high"
```

Les transitions sont déclarées grâce au mot "from" et décrivent comment passer de l'état 1 à l'état 2, après le mot "when" on déclare quels sensors doivent être actionnés pour effectuer la transitions (on aurait pu rajouter un "and "button2" becomes "high/low"" à la suite de cette exemple.

Dans Antlr, la transition à la forme suivante :

```
button is HIGH => off
```

Cela signifie que lorsque le bouton appelé 'button' est en position HIGH, on passe de l'état courant à l'état "off".

2.4 Déclarer l'état initial

Une fois toute les étapes précédentes terminées, on peut maintenant déclarer notre état initial.

```
initial "etat"
```

Cela a simplement pour effet de mettre le système arduino dans l'état "etat" au démarrage du système. Pour Antlr, il faut définir un état initial avec le symbole '->' avant le nom de l'état.

2.5 Déclarer les erreurs

Cette étape est optionnelle mais notre langage nous permet de déclarer des erreurs. Les erreurs sont des états du systèmes dans lesquels on reste bloqué jusqu'au redémarrage total du système.

```
error 8 when "button1" becomes "high" and "button2" becomes "high"
```

Pour les erreurs nous considéront admis que le montage dispose d'une led sur la broche 12, il ne faut donc pas que cette broche soit utilisée par un autre input / output.
On déclare notre erreur avec un entier, grace a cela notre erreur sera représentée par un cycle de clignotements correspondant à cet entier, par exemple avec "error 8", lorsque l'erreur sera déclencher, la led branchée sur la broche 12 clignotera 8 fois, attendra 2 secondes, puis clignotera 8 fois et ainsi de suite. Après avoir déclarer le numéro de l'erreur, on déclare le comportement qui va la déclencher avec "when", (le and est une fois de plus optionnel).

Dans Antlr, pour déclarer une erreur, on utilise la syntaxe suivante :

```
error 12 when button <= HIGH
```

Il a fallu implémenter la notion d'erreurs dans la grammaire d'Antlr, qui est composé d'un entier représentant le pin de la led dédié aux erreurs, et le fait d'une erreur possède une action (relié par le mot clé 'when').

2.6 Générer le code arduino

On doit terminer notre programme par :

```
export "name"
```

Cela a pour effet de générer le code arduino correspondant au programme que nous venons d'écrire. En Antlr, le code généré est affiché directement dans la console, l n'y a pas d'équivalent de la commande export.

3 Scénarios

Nous avons établi une liste de scénarios auxquels nos deux dsl peuvent répondre. Dans un premier temps, nous avons essayé de recréer les scénarios de base fournis dans le sujet, en implémentant de nouveaux mots-clés dans les langages, telles que le 'and' (pouvoir effectuer plusieurs actions en même temps), le 'or' (choix entre plusieurs actions). Nous avons ensuite mis en place la notion d'erreurs pour respecter la feature additionnelle.

Dans ce premier scénario nous allons voir comment allumer un buzzer et une led en appuyant sur un bouton, puis les éteindre en réappuyant. Voici le code à écrire pour Groovy :

```
sensor "button" pin 9
actuator "led" pin 10
actuator "buzzer" pin 11
state "on" means "led" becomes "high" and "buzzer" becomes "high"
state "off" means "led" becomes "low" and "buzzer" becomes "low"
initial "off"
from "on" to "off" when "button" becomes "low"
from "off" to "on" when "button" becomes "high"
export "Scenario1"
```

Et voici la version antlr :

```
sensor button: 9
actuator led: 12
actuator buzzer: 11
on {
    led <= HIGH
    buzzer <= HIGH
    button is LOW => off
}
-> off {
    led <= LOW
    buzzer <= LOW
    button is HIGH => on
}
```

Le second scénario décrit une alarme à double état, si on appuies sur 2 boutons simultanément, une alarme se délcenche, puis si on rappuies sur un des 2 boutons, l'alarme s'éteint.

```
sensor "button1" pin 9
sensor "button2" pin 10
actuator "buzzer" pin 11
state "on" means "buzzer" becomes "high"
state "off" means "buzzer" becomes "low"
initial "off"
from "on" to "off" when "button1" becomes "low" or "button2" becomes "low"
from "off" to "on" when "button2" becomes "high" and "button2" becomes "high"
export "Scenario2"
```

Version anltr :

```
sensor button1: 9
sensor button2: 10
actuator buzzer: 11
on {
    buzzer <= HIGH
    button1 is LOW or button2 is LOW => off
}
-> off {
    buzzer <= LOW
    button1 is HIGH and button2 is HIGH => on
}
```

Le 3eme scénario décrit une led qui s'allume lors d'un premier appuie sur un bouton, puis s'eteint lorsque l'on rappui.

```
sensor "button" pin 9
actuator "led" pin 10
state "on" means "led" becomes "high"
state "off" means "led" becomes "low"
initial "off"
from "on" to "off" when "button" becomes "high"
from "off" to "on" when "button" becomes "high"

export "Scenario3"
```

```
sensor button: 9
actuator led: 12
actuator buzzer: 11
on {
    led <= HIGH
    buzzer <= HIGH
    button is HIGH => off
}
-> off {
    led <= LOW
    buzzer <= LOW
    button is HIGH => on
}
```

Le 4eme scénario est une alarme à 3 états. Un premier appuie sur un bouton déclenche un buzzer, un second appuie arrête le buzzer et allume une led, un 3eme appui éteint la led, on retourne donc dans l'etat initial.

```
sensor "button" onPin 9
actuator "led" pin 10
actuator "buzzer" pin 11
state "ledstate" means "led" becomes "high" and "buzzer" becomes "low"
state "buzzstate" means "buzzer" becomes "high" and "led" becomes "low"
state "off" means led becomes low and "buzzer" becomes low
initial "off"
from "ledstate" to "buzzstate" when "button" becomes "high"
from "buzzstate" to "off" when "button" becomes "high"
from off to ledstate when button becomes high
export "Scenario4"
```

Version antlr :

```
sensor button: 9
actuator led: 12
actuator buzzer: 11

ledstate {
    led <= HIGH
    buzzer <= LOW
    button is HIGH => buzzstate
}
buzzstate {
    led <= LOW
    buzzer <= HIGH
    button is HIGH => off
}
-> off {
    led <= LOW
    buzzer <= LOW
    button is HIGH => ledstate
}
```

Et enfin le dernier scenario, qui montre un cas d'erreur : on possède 2 boutons et une led, si un des 2 boutons est actionné, la led s'allume normalement, si les 2 boutons sont actionnés en simultané, le système rentre dans une erreur, la led clignotera 3 fois puis marquera un temps d'arrêt avant de se remettre à clignoter.

```
sensor "button1" onPin 9
sensor "button2" onPin 10
actuator "led" pin 8
state "on" means "led" becomes "high"
state "off" means "led" becomes "low"
initial "off"
from "on" to "off" when "button1" becomes "high" or "button2" becomes "high"
from "off" to "on" when "button1" becomes "high" or "button2" becomes "high"
error 3 when "button1" becomes "high" and "button2" becomes "high"
export "ScenarioErreur"
```

Version antlr :

```
sensor button1: 9
sensor button2: 10
actuator led: 11
on {
    buzzer <= HIGH
    button1 is LOW or button2 is LOW => off
}
-> off {
    buzzer <= LOW
    button1 is HIGH and button2 is HIGH => on
}
error 3 when button1 <= HIGH or button2 <= HIGH
```

4 Analyse critique

Pour le choix du langage du DSL interne, nous étions au préalable parti sur Java, mais Groovy nous a finalement semblé plus adapté, notamment grâce à sa syntaxe moins contraignante et plus concise que celle de Java. Pour le choix du langage du DSL externe, ANTLR nous a semblé être un bon choix. Dans les 2 cas, ces langages font appel a un kernel écrit en java pour générer le code, le code DSL écrit par l'utilisateur va être interprété par le langage embarqué / externe et va appeler les fonctions du kernel java.
Pour ajouter la notion d'erreur (notre scénario bonus) nous avons créer un objet "Error" dans le kernel qui va contenir une liste d'actions, de sensors et de signaux. L'erreur ressemble fortement à un état mais elle ne possède aucune transition, c'est pour cela que lorsque l'on entre dans une erreur, on ne peut pas en sortir. Dans notre langage nous avons également du implémenter la notion de "and" et "or" afin d'avoir des conditions multiples pour pouvoir executer différents scénarios.Pour ce qui est de la génération de code d'erreur, l'erreur est appelée au debut de chaque état afin de vérifier si le système est dans un état d'erreur, voici un exemple d'une partie du code généré pour une erreur et 2 etats

```
void state_on() {
  state_error3();
  digitalWrite(8,HIGH);
  boolean guard = millis() - time > debounce;
  if( digitalRead(9) == HIGH && guard ) {
    time = millis();
    state_off();
  } else {
    state_on();
  }
}
void state_off() {
    state_error3();
  digitalWrite(8,LOW);
  boolean guard = millis() - time > debounce;
  if( digitalRead(9) == HIGH && guard ) {
    time = millis();
    state_on();
  } else {
    state_off();
  }
}
void state_error3() {
  boolean guard = millis() - time > debounce;
  if( digitalRead(9) == HIGH && digitalRead(10) == HIGH && guard ) {
    while true {
        for(i = 0; i < 3; i++){
            digitalWrite(12,LOW);
            delay(300);=
            digitalWrite(12, HIGH);=
        }
    }
  }
}
```

5 Répartition du travail

Matthieu s'est occupé de modifier le kernel et de développer le DSL interne (35%)
Michael s'est occupé de modifier le kernel et de développer le DSL externe (35%)
Mohamed a implémenté certaines fonctionnalités de groovy et d'antlr(30%)