Domain Specific Languages: ArduinoML

DOMAIN SPECIFIC LANGUAGES

BERTIN Loïc
FANTAUZZI Virgile
LADORME Guillaume
VIALE Stéphane

Sommaire

Introduction

Le but du projet était de développer deux langages (un interne et un externe) dans le but d'écrire du code compilable pour une Arduino. Nous avons développé le langage externe via MPS et le langage interne avec Groovy. Vous pouvez accéder à notre travail via le lien suivant: https://github.com/LoicBertin/Circular-DSL-Arduino

I - Description des langages développés

I.1) Schéma du domain-model

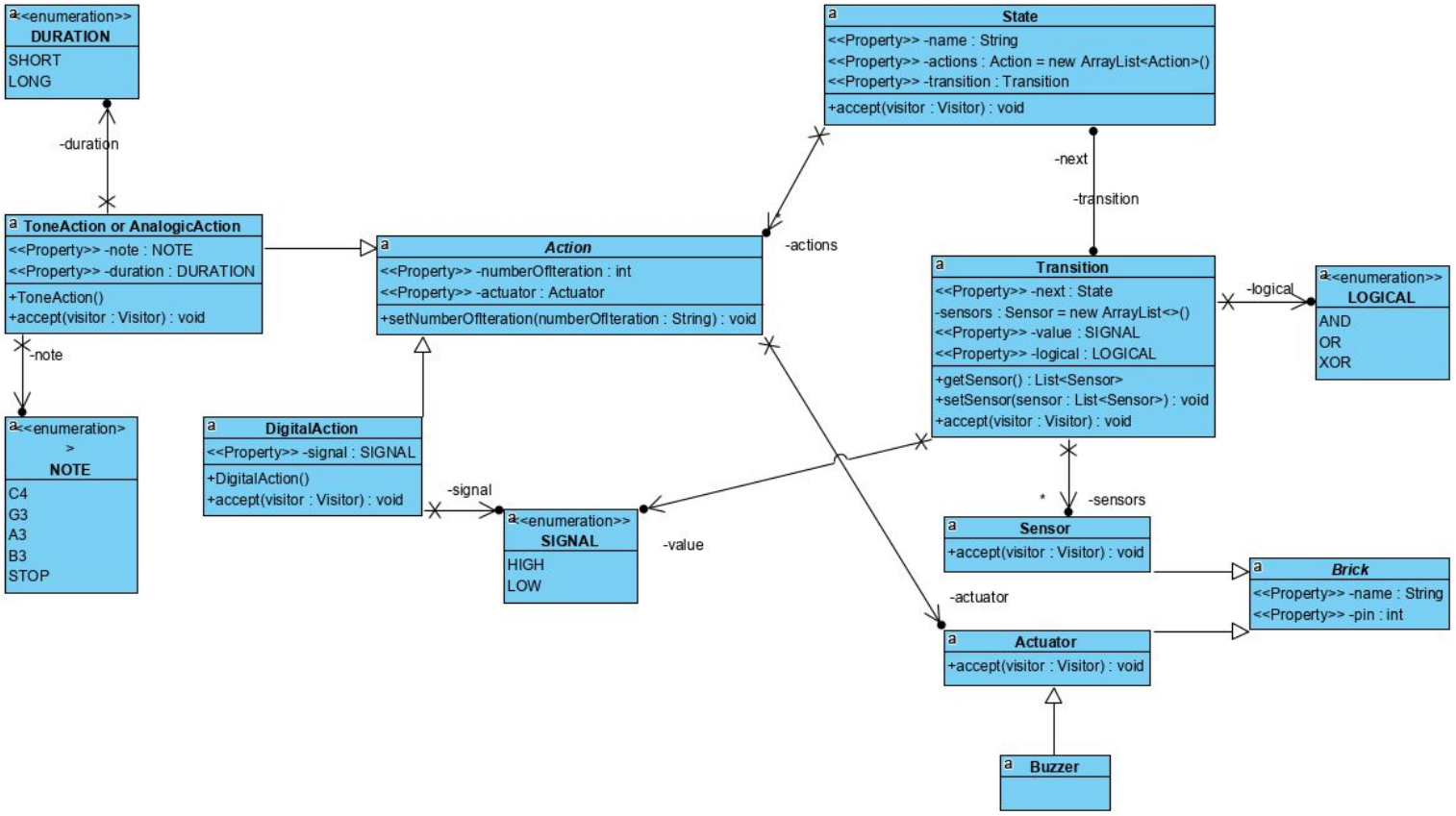

Diagramme de class du Domain Model

Voici le Domain Model utilisé dans le DSL interne, nous avons quelques différences mineures avec le DSL externe.

- Pas de classe Buzzer (un buzzer est un actuator, un sensor est un actuator et ils n'ont pas de différences de paramètres)
- La classe ToneAction s'appelle AnalogicAction
- Nous avons une DualTransition dans le state (il s'agit d'une seconde classe pour les transitions, mais nous reviendrons dessus plus tard)

I.2) Syntaxe du langage

<app (langage externe)> ::= <name (of the app)> “init_state :” <name (of the initial state)> {
<bricks>
<states>
}
<initial (langage interne)> ::= ”initial” <name (of the initial state)>
<bricks> ::= “bricks :”
(<sensor> | <actuator>)*
<sensor> ::= <name> <pin>
<name> ::= <letter>+
<pin> ::= <digit>+
<letter> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<actuator> ::= <name> <pin>
<states (langage externe)> ::= “states :”
(<name (of the state)> {
<action>*
<transition>
})*
<states (langage interne)> ::= “state” <name (of the state)> “means” <action>* “and”*
<transition>
<action> ::= <digitalAction> | <analogicAction>
<digitalAction> ::= <name (of an actuator)> “becomes ” <signal>
<signal> ::= “LOW” | “HIGH”
<analogicAction> ::= <name (of an actuator)> “plays” <note> “for” <duration>
<number_of_iterations> “time(s)”
<note> ::= “C4” | “G3” | “A3” | “B3”
<duration> ::= “short” | “long”
<number_of_iterations> ::= <digit>*
<transition (langage externe)> ::= <name (of a sensor)> “is” <signal> “=>” “<name (of a state)>”
<transition (langage interne)> ::= “from” <name (of the current state)> “to” <name (of the next state)> “when <name (of a sensor)> “becomes” <signal>

I.3) Description de notre extension et de comment nous l’avons développée

Nous avons choisi pour le langage interne et externe la même extension, il s’agit de “signaling stuff by using sounds”. Le but de cette extension est de pouvoir faire du morse par exemple en pouvant produire des sons courts ou longs et ainsi avoir un langage sonore pour les utilisateurs. Nos langages permettent pour un son de dire combien de fois on va le répéter, si c’est un son court (500ms) ou long (2000ms) et de quel son il s’agit (parmis une

liste de 4 sons mais qui est facilement enrichissable puisqu'il suffit de rajouter le code associé dans l'énumération NOTE). Pour cela il suffit d'écrire:
buzzer plays {nom_de_la_note} for {short, long} duration {nombre_d'itérations} time(s)

Pour réaliser cette extension nous avons donc ajouté une nouvelle transition pour les buzzers, appelée ToneTransition ou AnalogicTransition. En effet, pour les scénarios précédents, nous utilisions la transition par défaut et si nous lisions "HIGH" alors nous écrivions un tone avec une note à 255 et un temps fixe. et un "LOW" représentait un noTone. Cependant cette utilisation ne correspondait pas aux besoins de l'extension.
Cette nouvelle transition permet donc de lire une note définie par l'utilisateur et une durée pour cette note.
Pour le dsl interne il a donc fallu spécifier un nouveau mot clé pour différencier un sensor classique d'un buzzer. Le mot clé pour le sensor est "becomes" et celui pour le buzzer est "plays". De cette façon, nous pouvons différencier la transition que l'utilisateur veut utiliser et construire ensuite la bonne transition.

II - Scénarios implémentés

Nous avons implémenté chacun des scénarios de base avec un langage interne et un langage externe. Les scénarios étaient les suivant :

- **Very simple alarm:** Rester appuyé sur un bouton pour allumer une led et un buzzer, relâcher le bouton éteint la led et le buzzer.
- **Dual-check alarm:** Appuyer sur deux boutons simultanément déclenche un buzzer pendant que les boutons sont pressés et uniquement pendant qu'ils le sont.
- **State-based alarm:** Appuyer sur le bouton allume la led et la garde allumée tant que l'on appuie pas à nouveau sur le bouton.
- **Multi-state alarm:** Appuyer sur le bouton déclenche un buzzer, appuyer à nouveau arrête le buzzer et allume une led appuyer une troisième fois éteint la led et fait revenir dans l'état de départ.
- **Signaling stuff by using sounds:** Appuyer sur un bouton lance trois bips courts les uns après les autres. Appuyer à nouveau déclenche un bip long puis ramène à l'état de départ quand on relâche le bouton

III - Analyse de notre implémentation

III.1) ArduinoML

a) DSL Externe : MPS

Notre implémentation répond aux 4 scénarios de bases ainsi qu'à l'extension que nous avons choisie. En revanche, notre implémentation présente tout de même quelques faiblesses, notamment sur les composants d'éditions de MPS. En effet, si nous jouons le son STOP (qui arrête donc les sons infinis) nous sommes obligés de le renseigner de la façon suivante "buzzer plays STOP for {duration} duration {number_of_iterations} time(s)".

Or, il aurait fallu que l'utilisateur puisse écrire "buzzer plays STOP". De la même façon, si on joue une note pour "toujours" de la façon suivante "buzzer plays C4 for ever duration {number_of_iterations} time(s)" il aurait été mieux de pouvoir simplement écrire "buzzer plays C4 for ever". Pour régler ce problème, le jour du rendu nous avons trouvé qu'il était possible de faire des alternations dans les component editors de MPS et ainsi avoir plusieurs options en fonction des valeurs des paramètres d'une AnalogicAction et ainsi avoir un buzzer plus agréable à utiliser. Nous n'avons pas eu le temps de les mettre en place.
Une autre piste d'amélioration est notre Transition, en effet pour répondre au scénario 2 une DualTransition a été mise en place, mais il aurait été plus judicieux d'avoir une MultiTransition (qui serait le seul composant de transition) et qui pourrait avoir de 0 à N sensors ainsi que de 0 à N-1 LOGICAL pour pouvoir faire des règles logiques de transitions plus élaborées.

b) DSL Interne : Groovy

Le DSL interne répond aux besoins des scénarios de base ainsi que de l'extension.
Les points faibles de cette implémentation sont présents à 2 endroits :
Dans un premier temps dans le BaseScript de Groovy il y a une redondance de code dans le parser sur cette phrase : *from state1 to state2 when sensor [and/or sensor]*n becomes signal.*
Il y a une redondance, car en fonction du mot clé qui est lu (AND, OR ou XOR), on construit une transition différente. Il faudrait trouver un moyen de variabiliser cette information afin de la passer dans le constructeur.

Ensuite le second point faible se situe dans le kernel en java. Le toWiring de la classe Transition est devenu beaucoup trop complexe. Il aurait fallu faire de Transition une classe abstraite et faire des classes filles comme TransitionOr , TransitionAnd, etc.
Cela permettrait, dans l'immédiat, de rendre le code plus clair et pour l'avenir, de permettre de créer plus facilement de nouvelles transitions.

III.2) Technologies utilisées

Les technologies étant nouvelles pour nous, la prise de recul est assez limitée. Cependant après 1 mois d'utilisation nous avons déjà relevé certains points intéressants.

a) DSL Externe : MPS

Nous avons choisi d'utiliser MPS car il s'agit d'une technologie développée par JetBrains ce qui nous assure une maintenabilité, une documentation de qualité ainsi que des nouvelles features (si le projet devait continuer). Après avoir utilisé cette solution pendant le mois précédent, il se trouve que le développement via MPS n'est pas si facile que cela à prendre en main (heureusement les raccourcis d'autocomplétion sur les IDE de JetBrains facilitent le travail pour voir ce qui est possible de faire). Sans le tutoriel de M.Mosser il aurait été impossible de développer de but en blanc dans ce langage. Mais après avoir fait le tutoriel il est beaucoup plus simple de comprendre le fonctionnement de ce langage. Une fois le DSL développé, son utilisation est plutôt claire.

b) DSL Interne : Groovy

Nous avons choisi d'utiliser Groovy car le langage a été présenté durant le cours et qu'il est basé sur Java, une technologie que nous maîtrisons bien. Mise à part quelques modifications de paramètres à faire dans l'IDE pour ne pas avoir les lignes surlignées en rouge, Groovy a été relativement facile d'utilisation. Le code existant facilitant beaucoup la prise en main du DSL. La partie la plus complexe étant la prise en main du parser, une fois cette phase de compréhension passée, le développement est relativement fluide.

IV - Séparation du travail

Nous nous sommes séparés en deux groupes pour réaliser ce projet, un s'occupant du langage externe (Virgile et Stéphane) et un s'occupant du langage interne (Loïc et Guillaume).
Du côté du langage externe, Stéphane s'est occupé de réaliser le tutoriel de M.Mosser pour pouvoir comprendre le fonctionnement de MPS et avoir la base de code commune entre le langage externe et le kernel du langage interne (bien qu'après nous l'avons modifié pour qu'elle soit cohérente avec la pull request de M.Deantoni). Puis Virgile et Stéphane ont réalisé les scénarios et l'extension ensemble.
Du côté du langage interne, Loïc et Guillaume ont découvert Groovy et développé les scénarios ensemble en pair programming. L'extension du dsl interne a été réalisée par Loïc.
De plus, quand il fallait se mettre d'accord sur les éléments du langage à ajouter pour répondre aux scénarios nous discutions tous les 4 de la manière de procéder pour le faire afin d'avoir deux langages architecturé de la même façon.