Rapport du premier projet de DSL

Arduino ML

Par l'équipe C, composée de :

FERTALA Mohamed
ROCCELLI Valentin
BELKHIRI Abdelouhab
FARGEON Armand
EL ADLANI Rachid

Sommaire

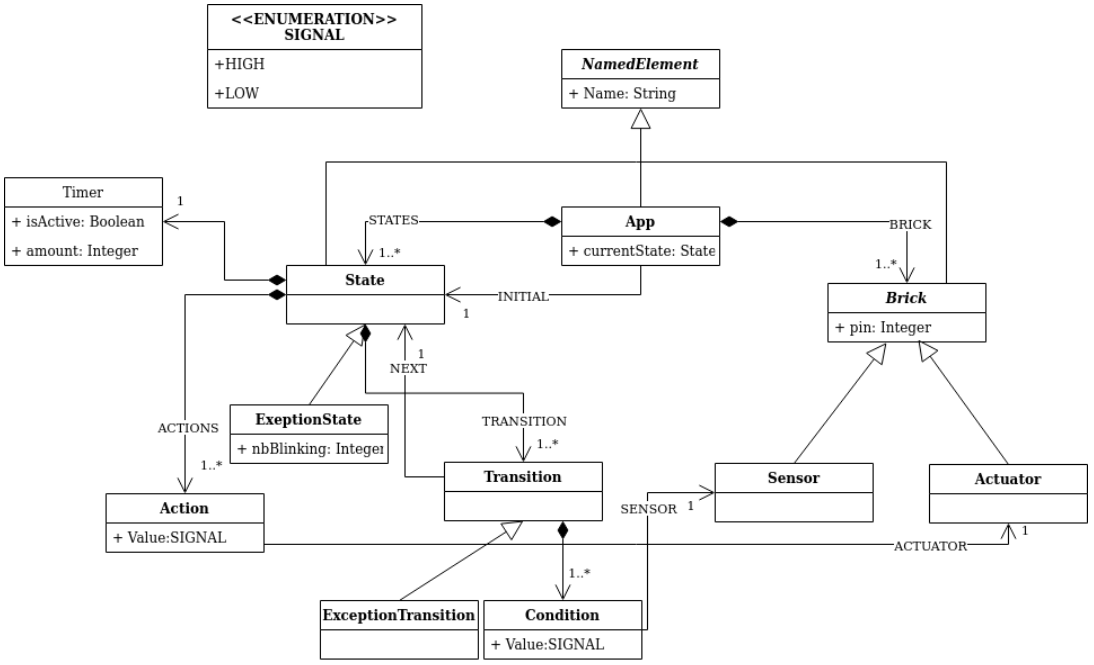
I - Lien vers notre code (dépôt GitHub)

https://github.com/RoccelliV/ArduinoML-kernel

II - Description des langages développés

II.A) Groovy

II.A.1) Le domain model sous la forme d'un diagramme de classe



II.A.2) La syntaxe concrète représentée sous une forme similaire au BNF

NOTE : (x)* veut dire que x peut être présent entre 0 et infini fois.

```
<app>  ::= bricks_definition states_definition states_details (waitFor)*
(error)* initial_state export

<bricks_definition> ::= (sensor | actuator) (sensor | actuator)*

<sensor> ::= “sensor” <string> “pin” <int> ‘\n’

<actuator> ::= “actuator” <string> “pin” <int> ‘\n’

<states_definition> ::= “states” <string> (“and” <string>)* ‘\n’

<states_details> ::= one_state_details (one_state_details)*

<one_state_details> ::= state (change_state)* ‘\n’

<state> ::= “state” <string> “means” element_changes ‘\n’

<element_changes> ::= <string> “becomes” <string>

<change_state>  ::=  “to”  <string>  “when”  element_changes  (“and”
element_changes)* ‘\n’

<waitFor> ::= “waitFor” <int>‘.’<string> “when” <string> ‘\n’

<error>  ::=  “error”  <string>  <string>  <int>  “on”  <string>  “when”
element_changes (‘and’ element_changes)* ‘\n’

<initial_state> ::= “initial” <string> ‘\n’

<export> ::= “export” <string>
```

II.A.3) Une description de nos extensions et comment elles ont été implémentées

Scénario 1, 3 et 4

Nous n'avons pas eu besoin de modifier le modèle de domaines pour les scénarios 1, 3 et 4. Nous avons uniquement changé la syntaxe du DSL afin qu'il soit plus maniable pour l'utilisateur.

Scénario 2

Nous avons ajouté l'objet ***Condition***, dont le cycle de vie est géré par l'objet ***Transition.*** L'objet ***Transition*** contient toutes les ***Condition***s nécessaires pour passer au ***State*** suivant.

Chaque objet ***Condition*** contient la valeur du ***Signal*** et le capteur associé au signal.

Nous avons également modifié le nombre de ***Transition*** dont un ***State*** est responsable pour le passer de seulement 1 à possiblement plusieurs. Ainsi, nous pouvons donc passer d'un ***State*** à, possiblement, plusieurs autres (e.g. : entre les states "**on**", "**off**" et "**one_btn_on**").

Scénario Temporal Transition

Pour répondre à ce scénario, nous avons créé 2 nouvelles classes, ***Timer*** qui est utilisé par la classe ***State***. Dans le DSL, nous allons avoir une façon de spécifier un temps d'attente lorsqu'une **brick** atteint un ***State*** spécifique.

```
waitFor 10.millisecond when "led" becomes "on"
```

On aura une sérialisation du mot "10.millisecond" en objet *Duration* qui utilisera un ***Enum*** qui contiendra une unité de temps relié à un multiplicateur (e.g *second("sec", 1000)*). Cela permettra de convertir le temps en milliseconde pour l'affecter à un objet ***Timer***, qui sera aussi affecté au ***State*** concerné qui sera responsable de son cycle de vie.

L'objet ***Timer*** fait partie du Kernel et il contient les attributs suivant :

- un booléen indiquant l'activité (ou l'inactivité) dudit Timer
- un entier représentant le temps que le timer doit attendre

Scénario Exception Throwing

Pour répondre à ce scénario, nous avons créé 2 nouvelles classes, ***ExceptionState*** et ***ExceptionTransition***, étendant respectivement les classes ***State*** et ***Transition***.

La classe ***ExceptionState*** contient en plus de la classe mère, un entier représentant le nombre de clignotements à faire avant de marquer une pause et de recommencer à clignoter comme décrit dans les "acceptances criterias". Il restera ainsi dans cet état d'erreur, dû à une violation de la règle interdisant l'événement de clicks sur 2 boutons simultanément.

Quand cet événement a lieu, une ***ExceptionTransition*** est créée. Cette transition bascule l'application de l'état actuel "sain" vers un état d'erreur : ***ExceptionState***. On détecte dynamiquement l'état actuel sans que l'utilisateur ne le renseigne afin de lui faciliter l'utilisation du DSL. Ainsi il n'aura pas à écrire tous les états qui peuvent mener à un état d'erreur s' il a défini de nombreux états dans son application.
L'exception se produit sur un ***capteur*** de type LED et clignote un nombre **x** de fois défini par l'utilisateur. L'état d'erreur ***ExceptionState*** aura comme nom "***error_x***".

II.B) Langium

La grammaire de Langium est très proche de celle de Xtext, même si nous avons dû effectuer quelques modifications pour porter la grammaire de base du projet Xtext (enum pas encore pris en charge dans la version 0.2.0, différences dans la gestion de l'héritage…) vers Langium.

II.B.1) Le domain model sous la forme d'un diagramme de classe

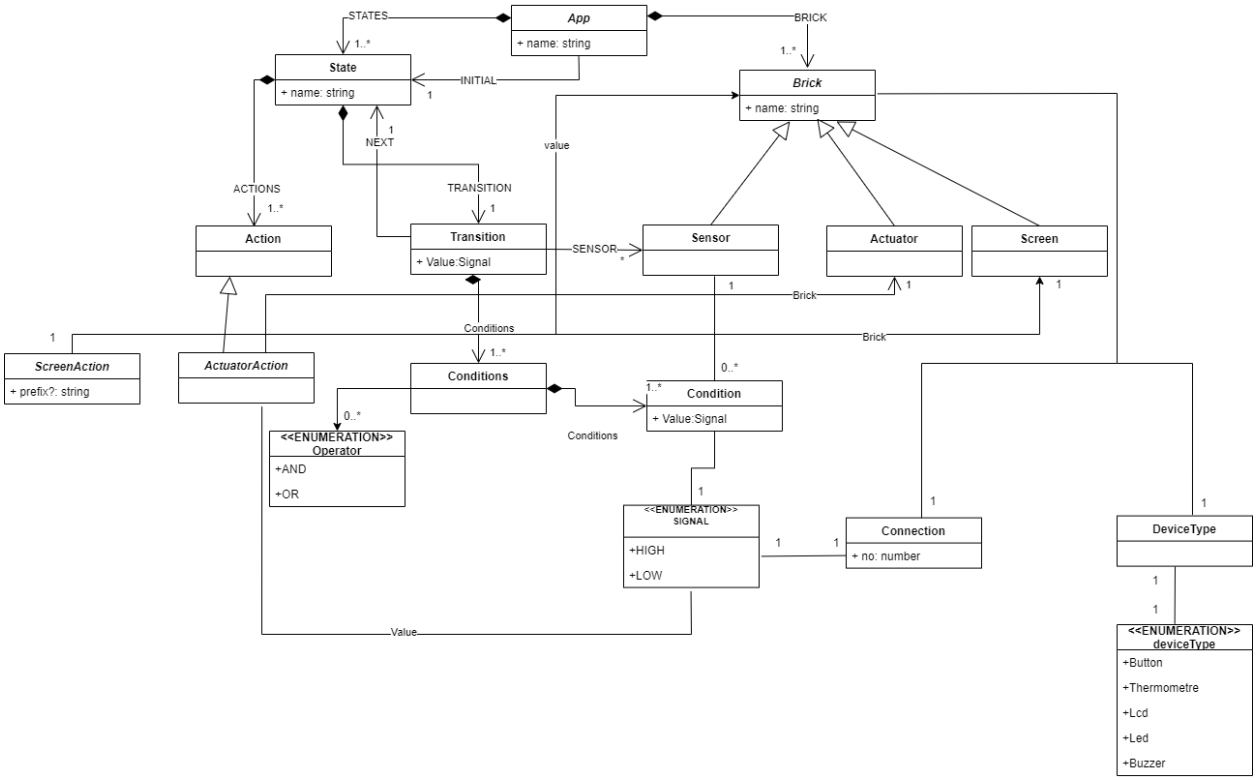


Image plus visible : [ici](#)

II.B.2) La syntaxe concrète représentée sous une forme similaire au BNF

NOTE : (x)* veut dire que x peut être présent entre 0 et infini fois.

```
<app> ::= app_init ‘{’ bricks states ‘}’

<app_init> ::= “app” <string> “initial state” <string>

<bricks> ::= “bricks” brick (brick)* ‘\n’

<brick> ::= ( “Button” | “Thermometer” | “Led” | “Buzzer” | “Lcd” ) <string> “: PIN” <int> ‘\n’

<states> ::= “states \n” state (state)*

<state> ::= <string> ‘{’ sensor_state (sensor_state)* change_state “}”

<sensor_state> ::= <string> “<=” <string> ‘\n’

<change_state> ::= actuator_state ( ( AND | OR ) actuator_state )* “=>” <string>

<actuator_state> ::= <string> “is” <string>
```

II.B.3) Une description de nos extensions et comment elles ont été implémentées

Tout comme pour le DSL Interne, le modèle de domaine de base n’a pas subi de modifications pour les scénarios 1, 3 et 4.

Concernant le scénario 2, plusieurs ajouts ont été nécessaires dans la grammaire. Les transitions d’un état à un autre ne permettait que d’indiquer l’état d’un ***Sensor*** pour enclencher le changement d’état.

Pour le scénario 2, nous allumons le buzzer dans le cas où deux boutons sont pressés en même temps. Pour ce faire, et pour simplifier l’écriture de ce scénario, les opérateurs OR et AND ont été ajoutés à la grammaire et une transition peut désormais prendre un ensemble de ***Sensor***s délimités par l’un de ces deux opérateurs dont l’évaluation permettra de passer ou non d’un état à un autre. Cela permet ainsi à l’utilisateur d’écrire ce type de transition :

```
button1 is HIGH AND button2 is HIGH => on
```

III - Les scénarios à la carte

III.A) Les scénarios du DSL Interne (*Groovy*)

Pour ce DSL, nous avons implémenté les fonctionnalités : “**Exception Throwing**” et “**Temporal Transitions**”.

Pour le premier scénario (celui validant la levée d’exceptions), nous initialisons 2 boutons et une LED. Nous allumons la LED si l’un des deux boutons est appuyé, nous l’éteignons dès que ledit bouton est relâché. Nous levons une erreur si les deux boutons sont appuyés en même temps. L’erreur se manifeste en faisant clignoter la LED de manière périodique (x clignotements avant d’attendre un moment et de nouveaux clignoter x fois) jusqu’à l’arrêt du système (ou un nouveau flash de programme).

Pour le second scénario (celui validant les transitions temporelles), nous initialisons 1 bouton et 1 LED. Lors d’un appui sur le bouton, nous passons dans l’état “on”. Lorsque nous rentrons dans cet état, nous attendons un certain temps avant d’allumer la LED.

De plus, nous avons mis un scénario simple qui échoue afin de mettre en évidence les *pins* déjà utilisés plusieurs fois par l’utilisateur.

III.B) Les scénarios du DSL Externe (*Langium*)

Pour ce DSL, nous avons implémenté la fonctionnalité “**Supporting the LCD screen**”.

Pour ce scénario, nous avons ajouté au modèle de domaine un nouveau type de **brick**, n’étant ni un actionneur, ni un capteur, mais simplement un écran. Nous nous sommes au départ posé la question de savoir si l’actionneur pouvait être utilisé pour implémenter l’écran, mais les connexions requises (un bus qui comprend plusieurs pin) et les actions sur ce nouvel appareil étaient trop éloignées de celles des autres types d’actionneurs. Un écran est donc déclaré avec un bus et dans les states, un actuator ou sensor peut être connecté à l’écran pour afficher son état. Selon l’actuator ou le sensor connecté, la valeur de l’appareil sera interprétée selon la nature du device.Si on connecte une led, elle sera affichée ON ou OFF. Si c’est un thermomètre, sa valeur sera transformée en degrés celsius.

Côté generator Langium, on va déclarer au début du fichier la dépendance requise pour utiliser l’écran, puis la lecture du device dont on veut afficher l’état sera lue et transmise à l’écran via la fonction print.

Un préfixe peut être ajouté pour afficher un texte avant l’affichage de l’état de l’appareil surveillé par l’écran.

IV - Analyse critique

IV.A) Prise de recul

IV.A.1) DSL interne - Groovy

IV.A.1.a) Remarques globales

Dans le cas de Groovy, nous avons pu voir une certaine flexibilité quant à l'implémentation des scénarios et d'autres fonctionnalités. Nous avons pu bénéficier du kernel de base avec la couche ***Groovy Model***, et ainsi pouvoir ajouter des attributs, ajouter des nouvelles classes en utilisant par exemple de l'héritage ou des implémentations d'interfaces assez facilement.

Groovy permet d'avoir une plus grande flexibilité par rapport à d'autres langages, on peut ainsi bénéficier de la métaprogrammation que l'on a pu expérimenter et mettre en place sur certains scénarios comme expliqué précédemment. La métaprogrammation nous a permis de mettre en place une meilleure visibilité dans le DSL selon nous en changeant par exemple la méta classe d'un type primitif : **number**.

Nous n'avons pas, dans un premier temps, rendu notre DSL très "user-friendly" car nous étions trop focalisé sur le fait de comprendre le mécanisme du DSL (models, kernel, basescript…).
Cependant, nous avons ensuite pu mettre en place des améliorations que nous allons décrire. Ceci permet à l'utilisateur d'avoir un maximum de "***services***". Nous sommes tout de même conscients que les services que nous avons mis en place sont limités. Le point principal à en tirer est que nous avons compris le principe qui en découle et nous pourrons le ré-utiliser pour le prochain projet.

IV.A.1.b) Détail des améliorations pour rendre notre DSL plus “user-friendly”

Gestion des exceptions : Pins déjà utilisées

Nous avons, en plus, procédé à quelques petites vérifications afin de permettre à l’utilisateur d’avoir des retour en cas d’erreur par exemple. Dans notre cas, nous avons une fonction permettant de vérifier si des **pins** ont déjà été utilisés et ainsi renvoyer une exception avec un message décrivant le type de “***brick***” et le numéro du **pin** qui fait défaut afin de permettre à l’utilisateur d’identifier clairement et rapidement son erreur.

```
executing scenario  UsedPinFail
Pins of type [sensor] - number : 10 already use
```

Gestion des exceptions : Méthode utilisée inexistantes

Nous avons également surcharger la méthode “***methodMissing”,*** ce qui permet à l’utilisateur dès lors qu’il utilise une méthode non existante, qui n’est donc pas prise en compte par notre DSL, de lui proposer une méthode qui s’apparente à celle-ci, en utilisant un algorithme (distance de levenshtein) pour s’en rapprocher.
Ainsi, une exception sera lancée afin de lui permettre d’avoir un retour sur la raison de l’erreur lors de la compilation. Dans le message qui est affiché il y aura le mot-clé que l’utilisateur a utilisé et qui ne correspond pas à notre grammaire, ainsi que le mot lui ressemblant le plus.

```
-------------------------------------------------------

Unable to find operation : tor

Did you mean the operation => to [arg0] ?

-------------------------------------------------------
```

Changement du format de base

Nous avons également procédé à un changement du format de base pour la déclaration des transitions entre états dans notre DSL. Désormais, il y a une définition des états que l’on souhaite utiliser pour un scénario quelconque. Cette dernière sera vérifiée pour éviter les états dupliqués. Nous avons retiré le “***from***” afin de permettre de spécifier, après la déclaration des états, leurs définitions et leurs transitions associées.
Exemple d’avant/après :

Avant	Après

`state "on" means "led" becomes "high"`
`state "off" means led becomes "low"`
 	
`from "on" to off" when "button" becomes high`
`from off" to "on" when "button" becomes high`
	`states "on" and "off"`
	
	`state "on" means "led" becomes "high"`
	`  to "off" when "button" becomes "high"`
	
	`state "off" means "led" becomes "low"`
	`  to "on" when "button" becomes "high"`

IV.A.2) DSL externe - Langium

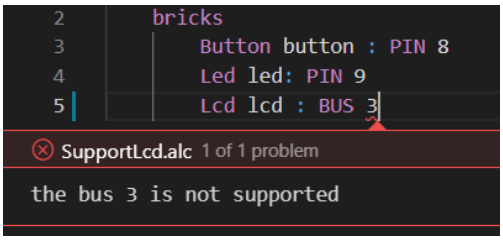
IV.A.2.a) Remarques globales

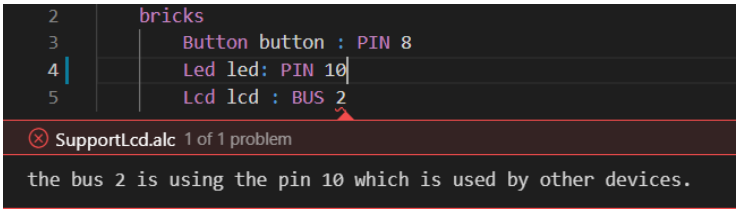
Langium nous permettait de reprendre le projet ArduinoML de base et de contrôler tous les aspects de notre nouveau projet Langium en utilisant du code du sous-projet Xtext et de bien comprendre chaque partie existante sous Xtext. Cette liberté nous a permis de comprendre plus facilement tout ce qu'on devait faire pour écrire ce DSL et de rapidement monter en compétences. Langium une fois pris en main, permet d'écrire très rapidement un DSL et également de l'enrichir au fur et à mesure. Le fait que Langium génère un **AST** réconcilié nous permet à chaque ajout de nouvelle entité dans la grammaire, de générer automatiquement sa représentation dans le typescript plutôt que de passer par un kernel intermédiaire et de devoir y faire des modifications.

IV.A.2.b) Détail des améliorations pour rendre notre DSL plus "user-friendly"

Vérifications statiques

Nous avons trouvé intéressant d'ajouter certaines vérifications statiques du code, pour prévenir l'utilisateur de certaines choses interdites dans son code. Si deux brick partagent le même pin, que ce soit via un bus ou un pin, la brick sera en erreur et le pin en faute sera indiqué à l'utilisateur. Pareil pour un pin ou bus non supporté.



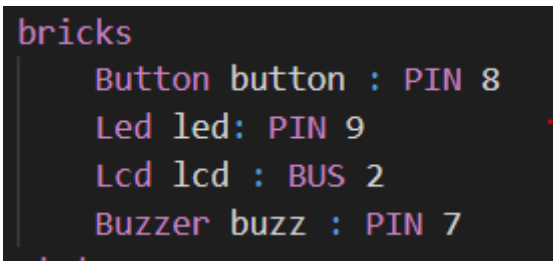


Résolution dynamique des pin des bus

Plutôt que de forcer l'utilisateur à préciser tous les pins du bus de la shield, on a mis en place dans le generator une translation dynamique du bus vers les numéros de pins correspondant. Il n'est pas pratique pour l'utilisateur de vérifier dans de la documentation arduino de la shield utilisé que le bus 2 correspond en fait aux pins de 10 à 16.

Types d'appareils supportés

Nous avons remplacé les termes Actuator et Sensor dans la déclaration des appareils. L'utilisateur déclare à la place pour chaque device, si c'est un bouton, un buzzer, un thermomètre ou encore une led. Il n'a plus besoin de savoir si c'est un actuator ou sensor, mais le langage pourra ensuite appliquer le traitement voulu aux différents appareils. Par exemple, s'il s'agit d'une led qui est déclarée, alors le DSL comprendra qu'il s'agit d'un actuator et que si elle est connectée à un écran, sa valeur HIGH correspond à ON et LOW à off.



IV.B) Les technologies que nous avons choisi d'utiliser

Nous avons choisi d'utiliser Groovy comme DSL Interne parce que le cours nous l'a super bien vendu. Il nous a paru bien et, après l'avoir essayé, nous l'avons tout de suite adopté, il était simple d'utilisation. Le seul regret que l'on pourrait avoir serait de ne pas avoir essayé les autres, mais, honnêtement, nous sommes très satisfaits de Groovy, nous n'avons eu aucun problème.

La prise en main est plutôt facilité par le fait qu'il ressemble à java et qu'il s'imbrique très bien avec le kernel lui écrit en java. Avec groovy nous n'avons pas eu besoin de connaître totalement le langage pour avoir des résultats rapidement.

Pour le DSL Externe, nous avons essayé ANTLR, XTEXT et Langium, avant de choisir ce dernier.
Nous nous étions d'abord orienté vers ANTLR. Après discussion avec le professeur nous avons choisi d'essayer des outils de "plus haut niveau", puisque ANTLR ne fournissait qu'un parser.
Nous allions donc essayer MPS, XTEXT et Langium. Après avoir rencontré des difficultés pour utiliser XTEXT avec Visual Studio Code et avec Eclipse. En effet, certaines dépendances du pom étant obsolètes, des clean packages ne fonctionnait plus et le fait de devoir utiliser Eclipse nous a également refroidis (habitué à Intellij depuis un moment maintenant), nous avons choisi de ne pas utiliser XTEXT.

Nous avons privilégié le test de Langium à celui de MPS car il semblait moins lourd et plus adapté à la création de PoC rapide (et ce fut le cas au vu des retours d'autres groupes).

Malgré le manque de documentation en ligne (qui est normal, l'outil étant TRÈS récent), nous avons trouvé l'utilisation de Langium assez facile et nous n'avons pas eu de problème majeur avec cet outil. Sa proximité avec Xtext au niveau de la syntaxe était également un gros plus. La deadline s'approchant dangereusement, nous avons alors choisi de l'utiliser et de ne pas tester MPS.
Comme pour Groovy, c'est dommage de ne pas avoir tout testé mais nous sommes satisfaits de notre choix.

V - La responsabilité de chaque membre du groupe dans ce projet

Noms	Actions faites
Armand **FARGEON**	Test de XTEXT sur Eclipse, implémentation du DSL Externe Langium.
Valentin **ROCCELLI**	Test de XTEXT sur VSCode, mise au propre du rapport.
Rachid **EL ADLANI**	Recherches sur les technos, Implémentation du DSL Interne Groovy.
Abdelouhab **BELKHIRI**	Recherches sur les technos, Implémentation du DSL Interne Groovy.
Mohamed **FERTALA**	Implémentation de la feature “à la carte” pour le DSL Externe Langium.