



# Rapport DSL - ArduinoML

## **Équipe F :**

Abbar Nabil

Assoti Lidaou Denis

Boukhiri Yassine

Oukpedjo Amirah

**2022 - 2023**

# Sommaire

<b>I. Introduction</b>	<b>2</b>
<b>II. Description du langage développé</b>	<b>3</b>
A. Modèle métier	3
B. Syntaxe concrète sous forme de BNF	4
1. DSL externe : Antlr	4
2. DSL interne : Python	5
C. Description des extensions réalisées	7
1. Remote Communication	7
2. Temporal transitions	8
3. Signaling stuff by using sounds	9
<b>III. Scénario implémentés</b>	<b>10</b>
SCÉNARIO 1 : Very simple alarm	10
SCÉNARIO 2 : Dual check alarm	11
SCÉNARIO 3 : State based alarm	12
SCÉNARIO 4: Multi state alarm	12
SCÉNARIO 5 : Temporal transition	12
SCÉNARIO 6 : Remote communication	13
SCÉNARIO 7: Signaling stuff by using sounds	14
<b>IV. Analyse critique de nos DSL pour le cas de l'ArduinoML</b>	<b>15</b>
A. Externe	15
B. Interne	16
<b>IV. Responsabilité des membres de l'équipe</b>	<b>17</b>
<b>V. Conclusion</b>	<b>18</b>

# I. Introduction

Dans le cadre de notre dernière année du cycle d'ingénieur en informatique à Polytech Nice-Sophia, un projet d'implémentation d'un langage dédié (en anglais, domain-specific language ou DSL) à la création des différents scripts arduino qui permettent le contrôle des différents composants arduino afin de répondre à des cas d'utilisations précis.

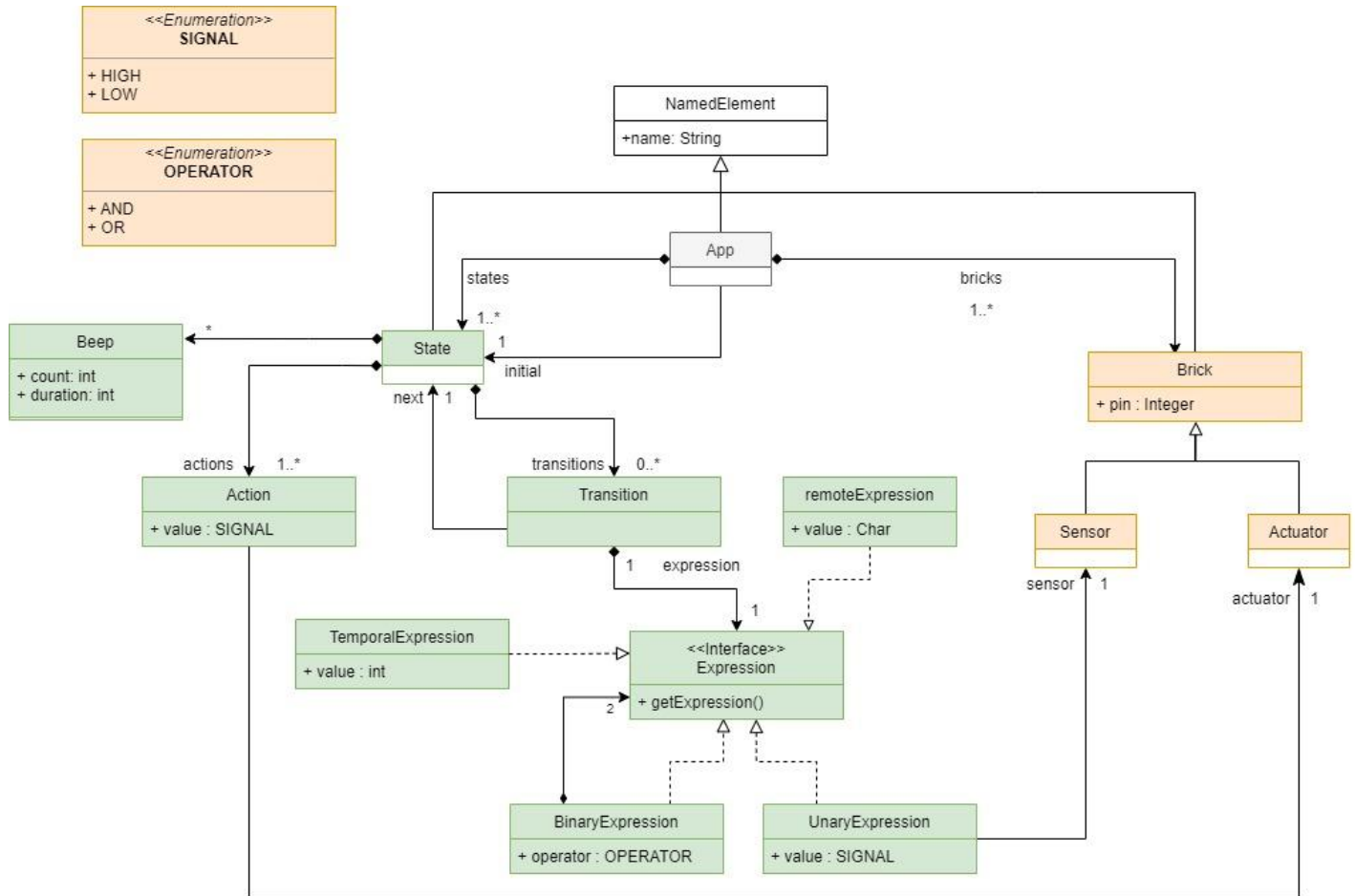
Afin de réussir ce projet, l'équipe devrait implémenter le DSL à la fois externe et interne. Le choix qui a été fait est : Antlr pour le DSL externe et Python pour le DSL interne.

Le but derrière le projet est d'apprendre les notions de base et la méthode d'implémenter des DSLs, externes et internes, en appliquant les connaissances acquises tout au long des séances de cours.

Lien vers repo : <https://github.com/denisassoti/si5-dsl-22-23-f-arduino>

## II. Description du langage développé

### A. Modèle métier



**Diagramme de classe - ArduinoML**

Le modèle métier présenté ici illustre tous les concepts pertinents pour le langage ArduinoML. Il est dérivé du modèle métier disponible sur le dépôt Zoo Architect, mais il a été amélioré pour prendre en compte les quatre scénarios de base dans un premier temps, puis les différentes extensions développées par la suite. Cette approche nous a donc permis de disposer d'un modèle métier adapté aux besoins du langage ArduinoML évolué.

Le modèle de base n'offrait pas la possibilité d'avoir plusieurs transitions de sortie au sein d'un même état; cette fonctionnalité a été prise en compte au sein de notre modèle métier par l'ajout d'une relation (0..n) entre les relations "State" et "Transition". De ce fait, dans notre langage on a la possibilité d'avoir des états terminaux (ou finaux).

Ayant plusieurs types de transitions et pouvant y avoir des compositions entre elles par des opérations (AND, OR), nous avons dû implémenter le **pattern composite** pour mieux gérer ce cas. De ce fait, chaque nouvelle expression dans notre langage sera insérée facilement. On a donc une transition de sortie qui est représentée par une seule expression (condition de sortie) qui elle-même peut-être une composition d'expression.

## B. Syntaxe concrète sous forme de BNF

### 1. DSL externe : Antlr

```

root      ::= declaration bricks states EOF

declaration ::= 'application' IDENTIFIER

bricks     ::= (sensor | actuator)+
sensor     ::= 'sensor' location
actuator   ::= 'actuator' location
location   ::= IDENTIFIER ':' PORT_NUMBER

states     ::= state+
state      ::= initial? IDENTIFIER '{' action+ (transition+)? '}'
action     ::= IDENTIFIER '<=' SIGNAL
transition ::= expression '=>' IDENTIFIER
initial    ::= '->'

expression ::= unaryExpression | '(' expression OPERATOR expression ')' | temporalExpression |
remoteExpression
remoteExpression ::= IDENTIFIER 'is' SIGNAL
unaryExpression ::= IDENTIFIER 'is' SIGNAL
temporalExpression ::= 'after' INTEGER 'ms'
remoteExpression ::= 'key' ALPHANUMERIC

PORT_NUMBER ::= [1-9] | '10' | '11' | '12'
INTEGER     ::= [0-9]+
IDENTIFIER  ::= LOWERCASE (LOWERCASE | UPPERCASE | DIGITS)+
SIGNAL      ::= 'HIGH' | 'LOW'
OPERATOR    ::= 'AND' | 'OR'
ALPHANUMERIC ::= [a-zA-Z0-9_]+

LOWERCASE   ::= [a-z]
UPPERCASE   ::= [A-Z]
DIGITS      ::= [0-9]

```

On retrouve à ce niveau les mêmes concepts que ceux présents dans notre modèle métier.

## 2. DSL interne : Python

```

<app> ::= <app_name> "=" <app_builder> "\\> <new_line> <app_content>
<app_name> ::= <identifier>
<app_builder> ::= "AppBuilder(\"> <identifier> "\")" <space>
<app_content> ::= <brick>+ <state>+ <get_content>
<brick> ::= <indent> (<sensor> | <actuator>)
<sensor> ::= ".sensor(\"> <location>
<actuator> ::= ".actuator(\"> <location>
<location> ::= <identifier> "\").on_pin(" <port> ") "\\> <new_line>
<port> ::= [1-9] | "10" | "11" | "12"

<state> ::= <indent> <state_name> <state_content>
<state_name> ::= ".state(\"> <identifier> "\")" "\\> <new_line>
<state_content> ::= <action>+ <beep>? <transition>
<action> ::= <indent> <indent> ".set(\"> <identifier> "\").to(" <high_low> ") "\\> <new_line>
<high_low> ::= "LOW" | "HIGH"

<beep> ::= <indent> <indent> ".beep(\"> <identifier> "\")" <beep_state>+ " "\\> <new_line>
<beep_state> ::= (<short_beep> | <long_beep>)
<short_beep> ::= ".short()" <beep_number>
<long_beep> ::= ".long()" <beep_number>
<beep_number> ::= ".times(" <non_zero_digit> ")"

<transition> ::= <terminal_transition> | <non_terminal_transition>+
<terminal_transition> ::= <indent> <indent> <no_transition> " "\\> <new_line>
<non_terminal_transition> ::= <indent> <indent> (<unary_transition> | <binary_transition> |
<remote_transition> | <temporal_transition>) <to_state> <new_line>
<unary_transition> ::= ".when(\"> <identifier> "\")" <has_value>
<binary_transition> ::= ".when()" <expression>
<expression> ::= ((<both> | <both_leaf>) (<and> | <and_leaf>)) | ((<either> | <either_leaf>) (<or> |
<or_leaf>))

<both> ::= ".both()" <expression>
<both_leaf> ::= ("> <identifier> "\")" <has_value>) | ("> (<remote_transition> |
<temporal_transition>))
<and> ::= ".and()" <expression>
<and_leaf> ::= ("> <identifier> "\")" <has_value>) | ("> (<remote_transition> |
<temporal_transition>))

<either> ::= ".either()" <expression>
<either_leaf> ::= ("> <identifier> "\")" <has_value>) | ("> (<remote_transition> |
<temporal_transition>))
<or> ::= ".or()" <expression>
<or_leaf> ::= ("> <identifier> "\")" <has_value>) | ("> (<remote_transition> |
<temporal_transition>))

<remote_transition> ::= ".key(\"> <letter> "\")"
<temporal_transition> ::= ".after(" <non_zero_digit> <digit>* ")"
<no_transition> ::= ".exit()"
<has_value> ::= ".has_value(" <high_low> ")"
<to_state> ::= ".go_to_state(\"> <identifier> "\")" "\\>

<get_content> ::= <indent> ".get_contents()"

<identifier> ::= <letter> ( <letter> | <digit> )*
<letter> ::= [a-z] | [A-Z]
<space> ::= " "
<new_line> ::= "\n"
<indent> ::= "\t"
<digit> ::= [0-9]
<non_zero_digit> ::= [1-9]

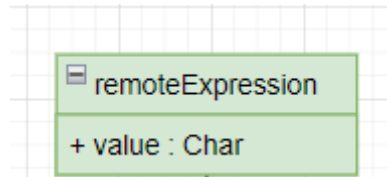
```

Afin d'avoir une idée plus claire des sorties qu'on peut avoir grâce à cette grammaire, veuillez visiter le lien suivant : [lien vers BNF interne](#). Ce dernier permet de générer aléatoirement quelques résultats qu'on peut avoir grâce à la grammaire proposée.

## C. Description des extensions réalisées

### 1. Remote Communication

Nous considérons ici que cette extension permet de définir des transitions de sortie d'un état vers un autre par l'intermédiaire d'une entrée au clavier. Par conséquent, nous avons défini un nouveau type d'expression, appelé "RemoteExpression", qui implémente l'interface "Expression". Nous avons jugé utile de permettre que ce type d'expression puisse être composé (associé à d'autres expressions par des opérateurs).



L'attribut "value" représente le caractère saisi par l'utilisateur.

#### Implémentation dans les DSL :

##### - Externe

La règle de grammaire est la suivante :

```

remoteExpression: 'key' key=ALPHANUMERIC;
  
```

Exemples d'utilisation :

**key 'e' => off**

**(button is HIGH AND key 'a') => on**

##### - Interne

Exemples d'utilisation :

```

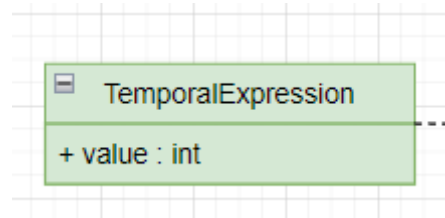
.when().key("e").go_to_state("off")
  
```

```

.when().both("BUTTON").has_value(HIGH).and_().key("a").go_to_state("on")
) \
  
```

## 2. Temporal transitions

Cette extension a été réalisée par la définition d'un nouveau type d'expression "TemporalExpression" qui implémente l'interface Expression. Ce type d'expression peut également être composé.



L'attribut "value" représente la durée pendant laquelle un état sera actif avant de passer à un autre état. Après l'expiration de cette durée, le système suivra la transition de sortie de l'état pour entrer dans un autre état.

### Implémentation dans les DSL :

#### - Externe

La règle de grammaire est la suivante :

```
temporalExpression : 'after' duration=INTEGER 'ms';
```

Exemples d'utilisation :

**after 800 ms => off**

**(button is LOW AND after 800 ms ) => on**

#### - Interne

```
.when().after(800).go_to_state("off") \
```

```
.when().both("BUTTON").has_value(LOW).and_().after(800).go_to_state("on") \
```



### 3. Signaling stuff by using sounds

Cette extension a été réalisée par la définition d'un nouveau concept (classe) "Beep" regroupant le nombre de beep à effectuer et l'intervalle d'exécution entre chaque beep. Ce concept étant intrinsèquement lié à un état, on a une relation de contenance entre le concept "State" et lui.



Comme choix de conception, on note qu'entre n'importe quels deux beeps la durée de silence est constante. En plus, les beeps sont des actions qui sont effectuées à l'entrée d'un état.

#### Implémentation dans les DSL :

##### - Interne

```
.beep("BUZZER").short().times(3) \
```

```
.beep("BUZZER").long().times(1).short().times(2) \
```

##### - Externe

Pas d'implémentation fournie.

### III. Scénario implémentés

#### SCÉNARIO 1 : Very simple alarm

En appuyant sur un bouton, on active une LED et un buzzer. En relâchant le bouton, les actionneurs sont éteints.

##### - Externe

```

application verySimpleAlarm

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
}

```

##### - Interne

```

app = AppBuilder("Scenario1") \
    .sensor("BUTTON").on_pin(9) \
    .actuator("LED").on_pin(12) \
    .actuator("BUZZER").on_pin(11) \
    .state("off") \
        .set("LED").to(LOW) \
        .set("BUZZER").to(LOW) \
        .when("BUTTON").has_value(HIGH).go_to_state("on") \
    .state("on") \
        .set("LED").to(HIGH) \
        .set("BUZZER").to(HIGH) \
        .when("BUTTON").has_value(LOW).go_to_state("off") \
    .get_contents()

```

## SCÉNARIO 2 : Dual check alarm

Une sonnerie est déclenchée si et seulement si deux boutons sont appuyés en même temps. Le fait de relâcher au moins un des boutons arrête la sonnerie.

### - Externe :

```
application dualCheckAlarm

sensor button1: 9
sensor button2: 10
actuator buzzer: 11

on {
    buzzer <= HIGH
    (button1 is LOW OR button2 is LOW) => off
}

-> off {
    buzzer <= LOW
    (button2 is HIGH AND button1 is HIGH) => on
}
```

### - Interne :

```
app = AppBuilder("Scenario2") \
    .sensor("BUTTON1").on_pin(8) \
    .sensor("BUTTON2").on_pin(9) \
    .actuator("LED").on_pin(12) \
    .actuator("BUZZER").on_pin(11) \
    .state("off") \
        .set("LED").to(LOW) \
        .set("BUZZER").to(LOW) \
        .when().both("BUTTON1").has_value(HIGH).and_("BUTTON2").has_value(HIGH).go_to_state("on") \
    .state("on") \
        .set("LED").to(HIGH) \
        .set("BUZZER").to(HIGH) \
        .when().either("BUTTON1").has_value(LOW).or_("BUTTON2").has_value(LOW).go_to_state("off") \
    .get_contents()
```

## SCÉNARIO 3 : State based alarm

En appuyant une fois sur le bouton, le système passe dans un mode où la LED est allumée.  
allumée. En appuyant à nouveau sur le bouton, la LED est éteinte.

### - Externe

```

application stateBasedALarm

sensor button: 9
actuator led: 12

on {
    led <= HIGH
    button is HIGH => off
}

-> off {
    led <= LOW
    button is HIGH => on
}

```

### - Interne

```

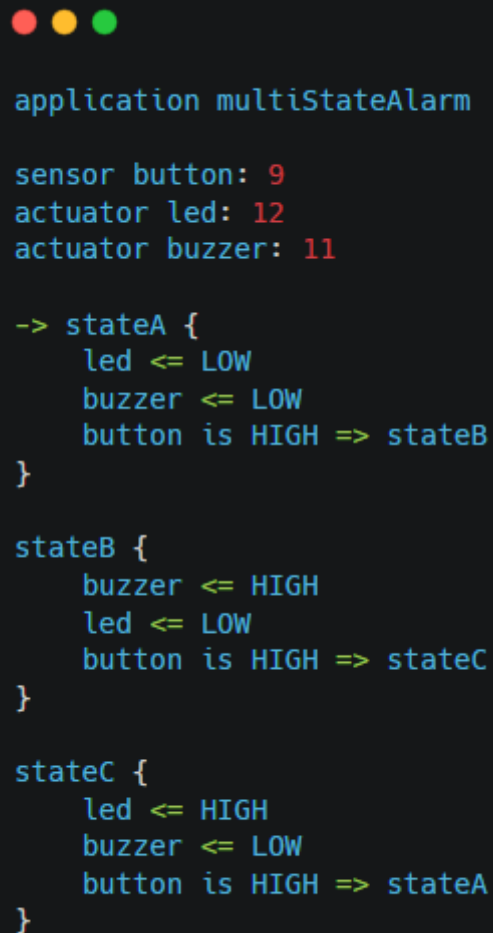
app = AppBuilder("Scenario3") \
    .sensor("BUTTON").on_pin(9) \
    .actuator("LED").on_pin(12) \
    .state("off") \
        .set("LED").to(LOW) \
        .when("BUTTON").has_value(HIGH).go_to_state("on") \
    .state("on") \
        .set("LED").to(HIGH) \
        .when("BUTTON").has_value(HIGH).go_to_state("off") \
    .get_contents()

```

## SCÉNARIO 4: Multi state alarm

En appuyant sur le bouton, le buzzer se déclenche. En appuyant à nouveau sur le bouton, le buzzer est arrêté et la LED s'allume. En appuyant à nouveau sur le bouton, la LED s'éteint et le système est prêt à faire du bruit après une nouvelle pression, et ainsi de suite.

- Externe



```
application multiStateAlarm

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

-> stateA {
    led <= LOW
    buzzer <= LOW
    button is HIGH => stateB
}

stateB {
    buzzer <= HIGH
    led <= LOW
    button is HIGH => stateC
}

stateC {
    led <= HIGH
    buzzer <= LOW
    button is HIGH => stateA
}
```

- Interne

```
app = AppBuilder("Scenario4") \
    .sensor("BUTTON").on_pin(9) \
    .actuator("LED").on_pin(12) \
    .actuator("BUZZER").on_pin(11) \
    .state("off") \
        .set("LED").to(LOW) \
        .set("BUZZER").to(LOW) \
        .when("BUTTON").has_value(HIGH).go_to_state("buzz") \
    .state("buzz") \
        .set("LED").to(LOW) \
        .set("BUZZER").to(HIGH) \
        .when("BUTTON").has_value(HIGH).go_to_state("led") \
    .state("led") \
        .set("LED").to(HIGH) \
        .set("BUZZER").to(LOW) \
        .when("BUTTON").has_value(HIGH).go_to_state("off") \
    .get_contents()
```

## SCÉNARIO 5 : Temporal transition

Alan veut définir une machine à états où la LED1 s'allume après une pression sur le bouton B1 et s'éteint 800ms plus tard, en attendant une nouvelle pression sur B1.

### - Externe

```

application temporalTransition

sensor button: 9
actuator led: 12

on {
    led <= HIGH
    after 800 ms => off
}

-> off {
    led <= LOW
    button is HIGH => on
}

```

### - Interne

```

app = AppBuilder("Scenario3") \
    .sensor("BUTTON").on_pin(9) \
    .actuator("LED").on_pin(12) \
    .state("off") \
        .set("LED").to(LOW) \
        .when("BUTTON").has_value(HIGH).go_to_state("on") \
    .state("on") \
        .set("LED").to(HIGH) \
        .when().after(800).go_to_state("off") \
    .get_contents()

```

## SCÉNARIO 6 : Remote communication

Alice veut définir une machine à états où la LED s'allume après une pression sur la touche 'a' du clavier et un appuie sur le bouton puis s'éteint après un appuie sur la touche 'e' du clavier.

### - Externe

```
application remoteCommunication

sensor button: 9
actuator led: 12

on {
    led <= HIGH
    key 'e' => off
}

-> off {
    led <= LOW
    (button is HIGH AND key 'a')=> on
}
```

### - Interne

```
app = AppBuilder("Scenario4") \
    .sensor("BUTTON").on_pin(9) \
    .actuator("LED").on_pin(12) \
    .state("off") \
    .set("LED").to(LOW) \
    .when().either().key("a").or_("BUTTON").has_value(HIGH).go_to_state("on") \
    .state("on") \
    .set("LED").to(HIGH) \
    .when().key("e").go_to_state("off") \
    .get_contents()
```



## SCÉNARIO 7: Signaling stuff by using sounds

Afin d'attirer l'attention de ses utilisateurs, Donald veut définir qu'une entrée dans un état émet trois bips courts consécutifs. De même, lorsque l'entrée dans un état signifie que le processus est terminé, Donald veut émettre un long bip.

```
app = AppBuilder("Scenario5") \
    .sensor("BUTTON").on_pin(9) \
    .actuator("LED").on_pin(12) \
    .actuator("BUZZER").on_pin(11) \
    .state("off") \
        .set("LED").to(LOW) \
        .beep("BUZZER").short().times(3) \
        .when("BUTTON").has_value(HIGH).go_to_state("on") \
    .state("on") \
        .set("LED").to(HIGH) \
        .beep("BUZZER").long().times(1) \
        .exit() \
    .get_contents()
```

## IV. Analyse critique de nos DSL pour le cas de l'ArduinoML

### A. Externe

La syntaxe du langage de notre DSL externe réalisée avec ANTLR4 est claire et intuitive, avec des constructions simples et faciles à comprendre pour un utilisateur expérimenté ou non en programmation Arduino.

Toutefois, nous n'avons pas pu fournir de services autour de notre DSL (éditeur web ou extension VSCode permettant la coloration syntaxique ou la validation d'erreurs); ce qui pourrait rendre l'utilisation de la DSL plus difficile pour les utilisateurs moins expérimentés. Il serait donc judicieux de mettre en place ces outils pour améliorer l'expérience utilisateur.

La validation quant à elle a été gérée au niveau du kernel.

Le processus actuel de génération de code avec notre DSL externe est le suivant :

- Créer et éditer un fichier ".arduinoml" contenant un scénario respectant notre grammaire ANTL4.
- Passer ce fichier en entrée de notre parseur ANTLR4 qui à son tour génère le code arduino. Cette opération se fait en ligne de commande maven; ce qui n'est pas pratique pour les utilisateurs non informaticiens.

Un moyen de remédier à cela serait de réaliser une API web au-dessus de notre parseur ANTLR4. Ce dernier grâce à une route REST prendra en entrée un fichier et après traitement renverra un fichier à l'utilisateur contenant le code arduino généré.

## B. Interne

Implémenter une DSL interne était plus ou moins facile et rapide à mettre en place. L'usage du pattern Builder et method chaining nous a permis de facilement construire notre langage après la définition du diagramme de classe.

La difficulté principale que l'équipe a rencontrée était de gérer les Binaryexpressions à l'aide du pattern composite. Mais finalement, l'équipe a pu surmonter ceci et à pu mettre en place un langage qui permet la composition de tout type d'expression : les "Unary expressions" qui lient des actionneurs à des états, des "Remote expressions" qui lient le clavier et les composants Arduino, et finalement les "Temporal expressions" qui lient le temps à des actionneurs.

L'expression utilisateur était toujours prise au sérieux par l'équipe qui voulait avoir un langage le plus intuitif aux utilisateurs du systèmes sans même qu'ils soient des informaticiens. Pour cela, le langage défini ressemble à une vraie langue parlée et n'importe quelle méthode prend un seul paramètre au maximum.

En plus des validateurs qui étaient déjà dans le projet initial, l'équipe a ajouté d'autres validateurs partout pour augmenter la sécurité du code généré. Par exemple pour les expressions binaires, on remonte à l'utilisateur des erreurs à chaque fois que le compilateur attend une méthode mais en reçoit une autre ou ne la reçoit pas. Également comme on a permis qu'une utilisation de beep par état (avec éventuellement un ou plusieurs short ou long beeps), on a ajouté un validateur pour empêcher également le fait de mettre plusieurs beeps.

## IV. Responsabilité des membres de l'équipe

- **Nabil ABBAR et Yassine BOUKHIRI**

Réalisation DSL interne python + mise en place des validateurs

- **Lidaou Denis ASSOTI et Amirah OUKPEDJO**

Réalisation DSL externe ANTLR4 + kernel java + expérimentation DSL interne avec java.

Tout le monde a participé à la rédaction du rapport et à la définition du modèle métier.

## V. Conclusion

L'équipe a pu suivre la démarche vu au cours pour implémenter une DSL interne et externe intuitive et facile pour les utilisateurs. Nous sommes fiers du travail dans le temps qui nous était alloué.

Ce projet en soi était une belle opportunité pour l'équipe pour découvrir les différentes manières d'implémenter les DSLs (interne et externe) et les appliquer afin de monter en compétences pour se préparer au prochain sujet.

Grâce aux diverses difficultés rencontrées au cours de la réalisation des différentes DSL, nous saurons mieux anticiper celles du prochain projet.