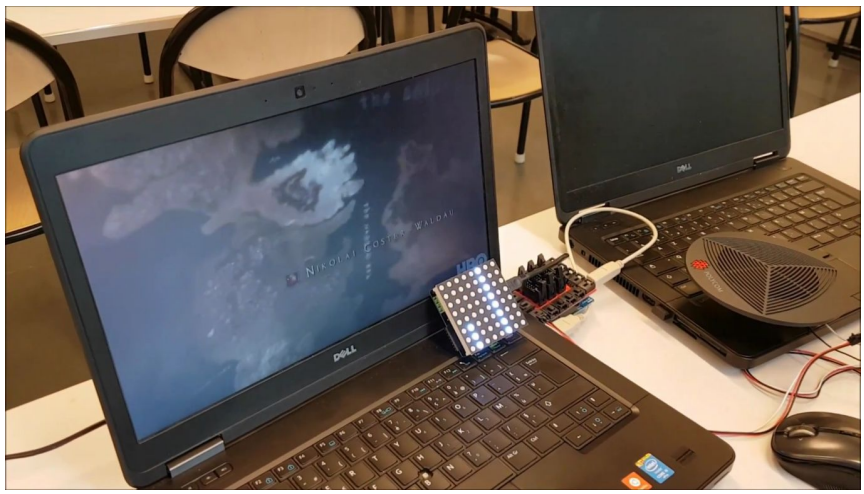
MusicalML

DSL

15/02/2017



Groupe **E**

Lisa JOANNO
Nicolas LECOURTOIS
Fabien VICENTE

Spécifications

Le sujet que nous avions originalement écrit est le suivant :

“Notre DSL s’adresse à des musiciens. Il lui permet de transcrire une partition grâce à des notes et des symboles musicaux. Ces symboles permettent de spécifier la durée des notes (noires, blanches…). Il permet de définir des macros afin de ne pas à avoir répéter une mélodie, telle qu’un refrain. Le code généré peut lire la musique dans le bon rythme et crée un affichage en temps réel sur un écran LED en fonction de la musique. “

En l’état actuel du projet, toutes les spécifications précédemment décrites ont été implémentées.

Choix du métamodèle & implémentation

Voici le processus que suit notre implémentation :

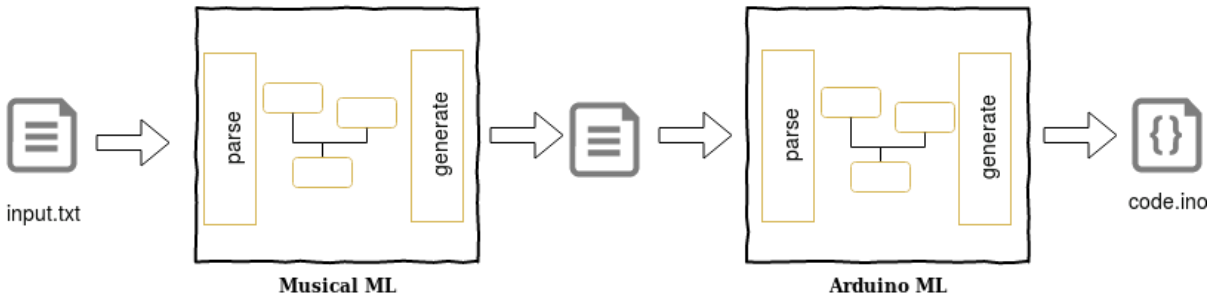


Figure 1 : processus d’interprétation

Nous avons donc deux métamodèles distincts dans notre projet : MusicalML, qui prend en entrée le fichier écrit en *Domain-Specific Language (DSL)* et qui produit en sortie un fichier de type machine à états, et ArduinoML, qui prend en entrée le fichier précédemment créé et qui produit un fichier téléversable sur une carte Arduino.

MusicalML

Le premier métamodèle est celui de la musique. Il prend en entrée le DSL écrit comme décrit dans la section Scénario utilisant le DSL. La grammaire utilisée par le parser est la suivante :

RuleSetGrammar.g4 ×

```
grammar RuleSetGrammar;

@header {...}

SYMBOL : '¤' | '#' | '=' ;
KEY_ALT : ('¤')* | ('#')* ;
NOTE : (NOTE_NAME | SILENCE) ;
COLOR : 'red' | 'green' | 'blue' | 'white' ;
NOTE_NAME : 'do' | 're' | 'mi' | 'fa' | 'sol' | 'la' | 'si' ;
SILENCE : '$' ;
CHOICE : 'HIGH' | 'LOW' ;
TEXT : LETTER+ ;
DIGIT : ('0'..'9')+ ;
LETTER : ('a'..'z' | 'A'..'Z')+ ;

init_color : 'color' COLOR ;
init_speaker : 'speaker' DIGIT ;
init_screen : 'screen' DIGIT ;
init_bpm : 'bpm' DIGIT ;
init_key : 'key' KEY_ALT ;
init_serial : 'serial' CHOICE ;

init : init_color init_speaker init_screen init_bpm init_key init_serial ;
macro_def : '-' TEXT '-' '{' note+ '}' ;
note : SYMBOL? NOTE DIGIT? ('+' | '-')* '.'? ;
score : (note | '-' TEXT '-' | macro_def)+ ;
dsl : init macro_def* 'score' score ;
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ -> skip ;

```

Nous avons plusieurs notions élémentaires dans cette grammaire :

- Les initialisations :
    - La couleur potentiellement utilisée pour l'écran LCD
    - Le PIN utilisé pour brancher le haut-parleur
    - Le PIN utilisé pour brancher le LCD
    - Le BPM, le tempo
    - L'altération à la clé
    - L'activation ou non du LCD
- Une partition, qui contient une suite de notes et de noms de macro
- Une macro, qui est un groupe de notes qui se répètent, comme un refrain
- Une note, qui est définie par sa hauteur, son altération et sa durée
- Une note peut également être un silence, donc un son *nul*, qui n'est caractérisé que par la durée du silence

Le métamodèle correspond point par point à ces notions élémentaires :

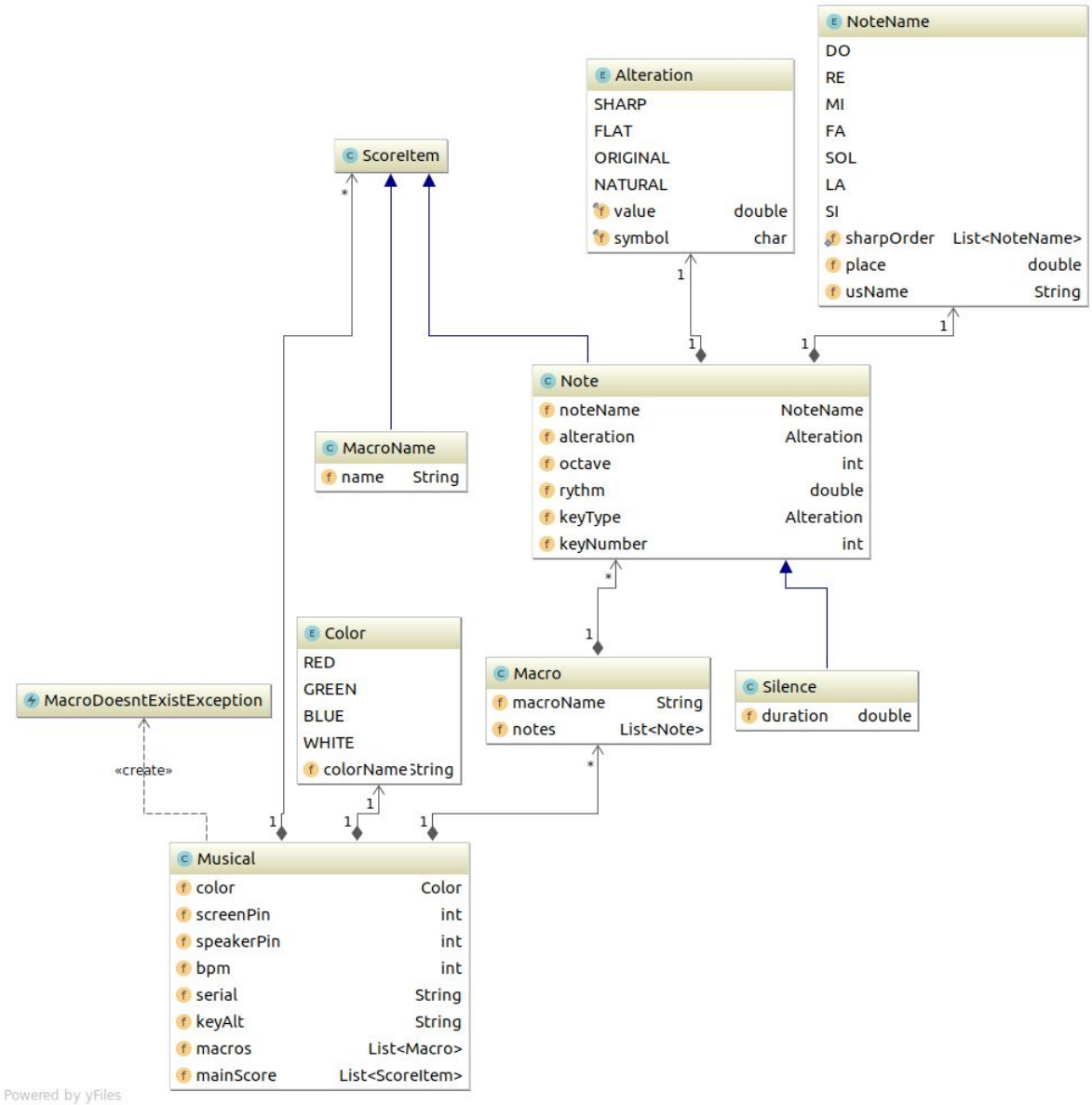


Figure 2 : métamodèle de MusicalML

L'idée originale était de lire les macros, de stocker les notes qu'elle contient, de commencer la partition, et pour chaque référence à une macro, on allait chercher les notes de celles-ci et on les plaçait à la place de la macro.

Cependant, on utilise les listeners d'Antlr4. La propriété de ces listeners est qu'ils parcourent les règles reconnues une par une, sans notion d'ordre. Chaque règle reconnue dans le fichier d'entrée est traitée séparément : la définition des macros avant la partition est donc potentiellement faite séparément de la définition de la partition. Or, dans la partition, l'utilisateur utilise le nom des macros qu'il a déclaré. Donc, quand dans la partition l'utilisateur écrit "-refrain-", la macro "refrain" n'est pas forcément déjà créée dans la classe *Musical*.

C'est pour cela qu'on utilise un concept de *ScoreItem*. Un ScoreItem est soit le nom d'une macro, soit une note. On remplit chaque macro quand elle est définie, on remplit la partition par des *ScoreItem* qui seront des *Note*, et quand on rencontre une macro, on ajoute un *ScoreItem* de type *MacroName*. Cette implémentation évite d'avoir à parcourir deux fois le fichier.

Lors de la génération du fichier (quand on imprime le fichier), le modèle est 100% rempli, nous avons juste à appeler le *toString()* des *ScoreItem* de la partition, et de définir celui-ci pour la classe *Note*.

Lors de la génération du fichier d'output de MusicalML, on imprime donc juste les initialisations et chaque *Note*. Ce qui donne :

Input de MusicalML — 1

```
color blue
speaker 8
screen 12
bpm 360
key ##
serial LOW

score
mi2
```

Output de MusicalML / Input d'ArduinoML — 2

```
actuator led : 12
speaker spk : 8
init: s0
serial LOW
debounce : 100

state s0 {
    serialPrint zb
    when 0 ms elapsed => s1
}state s1 {
    serialPrint e166
    led <= HIGH
    tone spk <= 165 hz for 166 ms
    when 215 ms elapsed => s2
}

state s2 {
}
```

Output d'ArduinoML — 3

```
void setup() {
    pinMode(12,OUTPUT);
    pinMode(8, OUTPUT);
}
bool acted = false;long time = 0; long debounce = 100;
void state_s0(){
    if(!acted){
        acted = true;
    }
boolean guard = millis() - time > debounce;
    if(    (millis() - time) >= 0  && guard){
        acted =false;time = millis();state_s1();
    }else {
        state_s0();
    }
}
void state_s1(){
    if(!acted){
    digitalWrite(12,HIGH);
    tone(8,165,166);
        acted = true;
    }
    boolean guard = millis() - time > debounce;
    if(    (millis() - time) >= 215  && guard){
        acted =false;time = millis();state_s2();
    }else {
        state_s1();
    }
}
void state_s2(){
    if(!acted){
        acted = true;
    }
    boolean guard = millis() - time > debounce;
}
void loop() { state_s0(); }
```

ArduinoML

Le second métamodèle en est un plus généraliste, conçu pour répondre aux spécifications du kernel ArduinoML. En voici la grammaire :

RuleSetGrammar.g4 ×

```
1   grammar RuleSetGrammar;
2
3   @header {
4       package grammar;
5   }
6
7   dsl : (actuator | sensor | speaker)+ init+ serial? debounce? state* ;
8
9   debounce: 'debounce : ' DIGIT;
10  sensor : 'sensor' TEXT ':' DIGIT;
11  actuator : 'actuator' TEXT ':' DIGIT;
12  speaker : 'speaker' TEXT ':' DIGIT;
13  serial : 'serial' binaryState;
14  init : 'init:' TEXT;
15  action : (logicalAction | tone | serialPrint);
16  serialPrint : 'serialPrint' TEXT;
17  logicalAction : TEXT '<=' binaryState;
18  tone : 'tone' TEXT '<=' DIGIT ('hz'|'Hz') 'for' DIGIT 'ms';
19
20  condition : (timeCondition | logicalCondition);
21  timeCondition : DIGIT 'ms' 'elapsed';
22  logicalCondition : TEXT 'is' binaryState;
23  transition :'when' condition ('and' condition)* '=>' TEXT;
24  state : 'state' TEXT '{' action* transition*'}';
25  binaryState : ('HIGH' | 'LOW');
26
27  DIGIT: ('1'..'9')? ('0'..'9')+;
28  WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ -> skip ;
29  TEXT : LETTER+ ;
30  LETTER : ('a'..'z' | 'A'..'Z' | '0'..'9')+ ;
31
```

Figure 3: Grammaire de ArduinoML

Le code n°2 est un exemple de code reconnu par cette grammaire. Le choix a été fait de garder une grammaire semblable à celle utilisée dans le noyau MPS. Celle-ci nous semblait adaptée à une machine à état. À celle-ci, plusieurs extensions ont été ajoutées :

- Les conditions multiples afin de combiner (via un *and*) les conditions
- Les conditions temporelles afin de changer d'état au bout d'un certain temps
- L'action *serialPrint* qui permet à l'utilisateur d'écrire sur le port série
- L'action *tone* pour pouvoir jouer un son à une fréquence donnée pendant un temps donné

À cela nous avons aussi ajouté des valeurs d'initialisations et la brique *speaker* afin de ne pas confondre un actionneur logique classique (comme une led) et un haut parleur que nous utilisons différemment. Avec ces modifications nous gardons donc toutes les capacités du ArduinoML originel. Pourtant celles-ci ont pour but de rendre possible l'implémentation des fonctionnalités de MusicalML. Par exemple, la condition

temporelle permet d'attendre la fin de la note sans jouer la suivante, *sensor* permet de jouer la note, *serialPrint* permet lui de parler à l'écran. Toutes ces fonctionnalités ont pour point commun de rester bas niveau, ceci a une raison : le but est d'éviter les fuites d'abstractions. Alors que MusicalML est dédié à la musique, ArduinoML doit rester générique, ArduinoML sait parler en série mais ne connait pas l'écran.

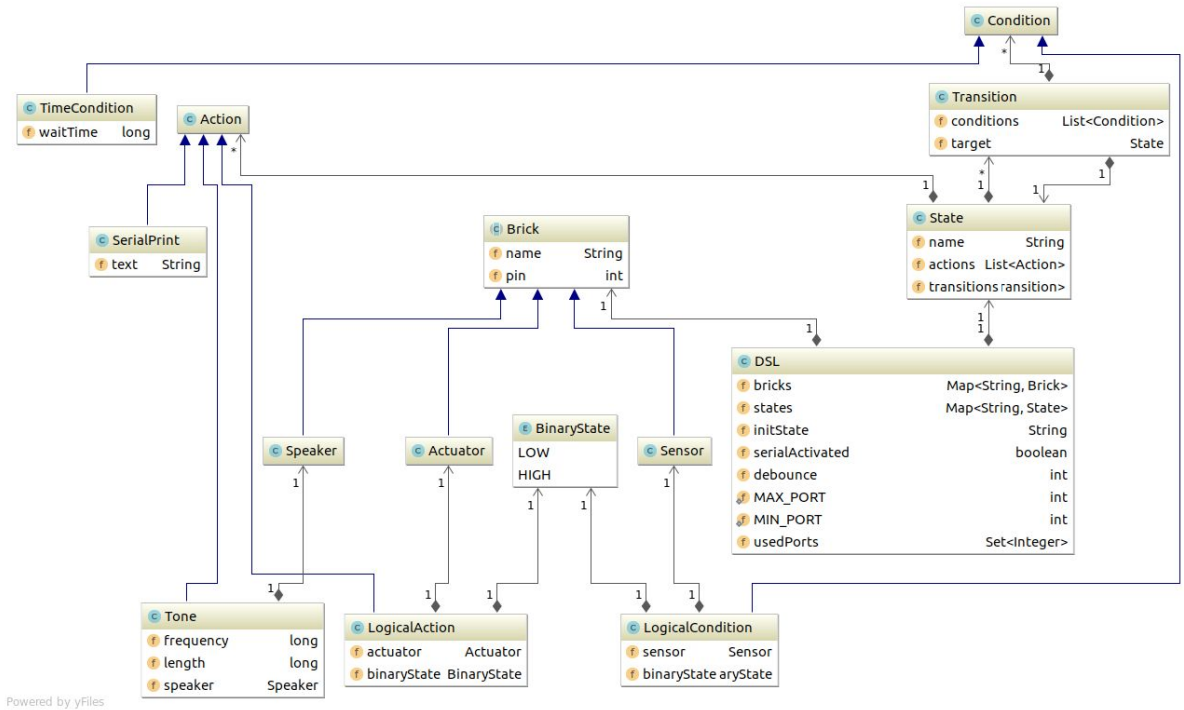


Figure 4: Métamodèle de ArduinoML

De la même façon, le métamodèle, inspiré du cookbook MPS, reste proche de celui de base. Les modifications principales apportées correspondent aux nouveaux types d'éléments ajoutés. Le schéma de base (état, transitions et briques) reste le même et devient juste enrichi de nouveaux éléments apportant de nouvelles fonctionnalités.

La génération du code elle aussi est une version enrichie de celle du modèle "automate" du kernel a un détail prêt. Une vérification est faite qu'il y a bien eu un changement d'état avant d'effectuer les actions liés à celui-ci, ce qui ne change rien pour un actioneur classique (une led mise à HIGH ou LOW garde son état par exemple) mais permet le bon fonctionnement de serialPrint et Tone qui eux ne doivent être appelés qu'une fois par changement d'état.

Cette génération de code(voir le code 3) nous permet d'atteindre nos objectifs mais a un désavantage. En gardant ce format de code en machine à état nous générons du code qui n'est pas nécessaire pour MusicalML, ceci n'est pas un problème en soit, mais si nous arrivons à un grand nombre de notes (plus de 140 environ) nous générons un code qui, une fois compilé, ne rentrera pas dans la mémoire interne d'un Arduino. Malgré tout, nous pensons que ce choix reste le bon pour éviter de spécialiser inutilement notre langage.

Scénario utilisant le DSL

MusicalML est destiné aux musiciens. Gilbert fait partie d'un orchestre et, pour leur prochain concert, on lui a demandé d'apprendre à jouer le thème principal de la marche impériale à la flûte à bec. Il a trouvé une partition qui le lui enseignerait, mais il apprécierait davantage de l'entendre. Il trouve également intéressante la possibilité que lui offre MusicalML d'avoir un repère visuel rythmique de sa musique.

Gilbert récupère se munit de sa partition, d'une carte Arduino, d'un haut parleur et d'un écran à LED. Il connaît la musique, et sait lire sur la partition les informations dont il a besoin. Le fichier qu'il écrit en MusicalML est composé de deux parties. Une première où il spécifie les paramètres nécessaires liés à son haut-parleur et son écran LED et les méta-informations liées à sa musique comme son tempo, et ses altérations, et une seconde qui est la retranscription de sa mélodie. Pour l'écrire, il se rend sur https://github.com/lisajoanno/DSLbyANTLR et suit les instructions du README.

Celui-ci lui le guidera depuis la retranscription de sa partition jusqu'au téléversement du code Arduino en passant par l'étape d'exécution de notre script de génération.

Analyse de la technologie utilisée

La principale difficulté que nous avons rencontrée pendant le développement de ce projet est le type de la grammaire d'Antlr : LL(1).

Ce type de grammaire est ultra-performant, et il permet de d'avoir rapidement un résultat pour une DSL, mais il est limité sur les possibilités de reconnaissance. Par exemple, il nous empêche de laisser à l'utilisateur le choix de l'ordre dans lequel il définit les paramètres (bpm, key, screen, …). En cela, il est différent d'une grammaire comme celle de Yacc, qui est LALR, et qui permet plus de flexibilité lors de son écriture, mais qui est plus lente.

Notre choix s'est avéré être le bon. Nous avons concentré notre travail sur l'implémentation des règles, la définition de la grammaire et non pas sur la configuration de Antlr. C'est un outil efficace et productif dont nous ne regrettons pas le choix.

Répartition des tâches

Lisa : grammaire et métamodèle de MusicalML + mise en place de l'environnement de développement Antlr4.
Nicolas : spécifications du code Arduino à générer par ArduinoML + implémentation des problématiques de l'expert pour MusicalML.
Fabien : grammaire et métamodèle de ArduinioML + protocole de sérialisation pour l'écran LCD.