Université de Nice - Sophia Antipolis

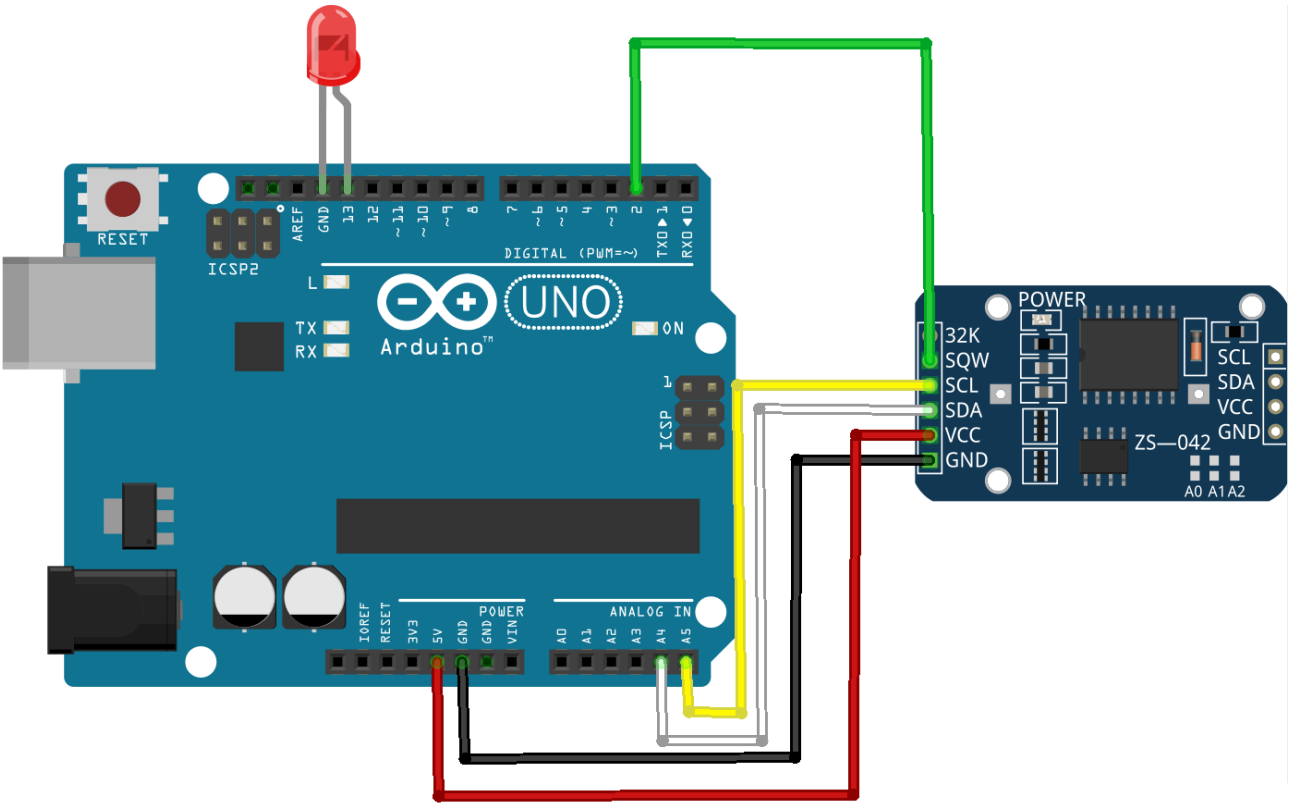
Polytech Nice Sophia

Sciences Informatiques 5ème année

Domain Specific Language

ArduinoML

Florian Juroszek, Théo Foray, Laura Lopez



Enseignant responsable : Julien Deantoni

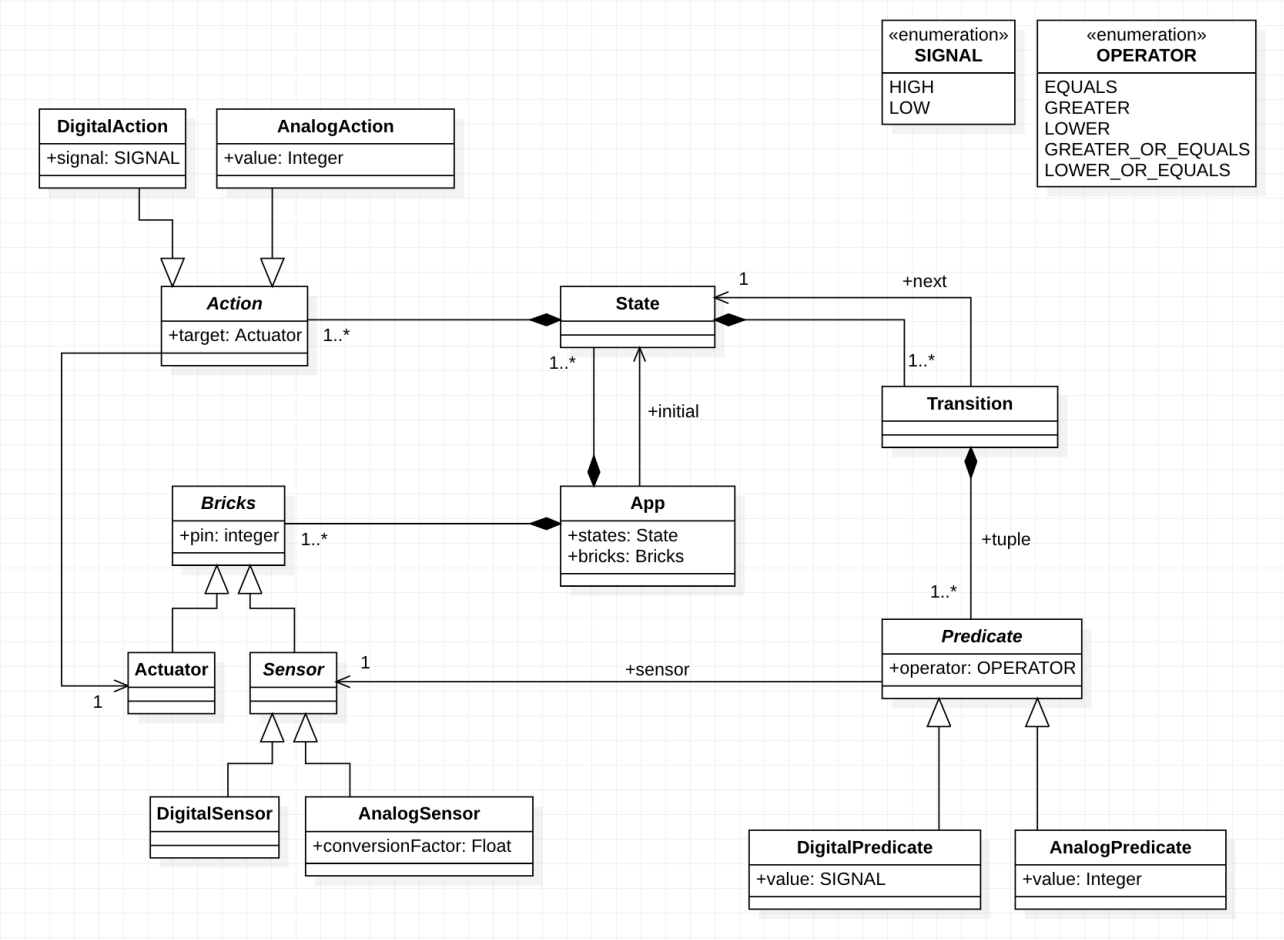
Table des matières

1 Informations

Le code de nos DSLs (en Groovy et avec MPS) est disponible sur GitHub à l'adresse suivante : https://github.com/FlorianJuroszek/dsl-arduinoml.

2 Notre langage

2.1 Le modèle sémantique



Nous avons, avant chaque développement, mis à jour notre *semantic model* afin de s'assurer d'une part de la cohérence de nos choix et d'autre part de pouvoir se partager au sein de l'équipe un document de référence sur la modélisation attendue. Nous avons utilisé le logiciel StarUML [1] qui sauvegarde le diagramme sous forme de fichier json permettant d'être facilement versionné.

1. http ://staruml.io

2.2 Syntaxe

2.2.1 DSL externe

Dans MPS, la syntaxe est formalisée dans les fichiers *Editors*. Voici une traduction "*BNF like*" de cette syntaxe :

```
<App> ::= "application: " <string> "bricks": <Brick>+ "states:" <State>+ "initial state: " <string>
```

Brick est un concept abstrait et n'a donc pas de syntaxe directe, et le concept *Bric* dans *App* utilise donc les sous-classes de *Brick*. De plus, la *string* utilisée après "*initial state*" se doit de correspondre au nom d'un des états.

```
<CommonBrickProperties> ::= <string> " on pin " <int>
<Actuator> ::= "actuator " <CommonBrickProperties>
<AnalogSensor> ::= "analog sensor " <CommonBrickProperties> " with factor " <float>
<DigitalSensor> ::= "digital sensor " <CommonBrickProperties>
<State> ::= "state " <string> <Action>+ <Transition>+
```

Action est un concept abstrait et n'a donc pas de syntaxe directe, et le concept *Action* dans *State* utilise donc les sous-classes *d'Action*.

```
<CommonActionProperties> ::= <string>
```

De la même manière que pour "*initial state*" dans l'App, la *string* dans *CommonActionProperties* se doit d'être le nom d'un des *Actuator*.

```
<AnalogAction> ::= <CommonActionProperties> " takes " <int>
<DigitalAction> ::= <CommonActionProperties> " becomes " <SIGNAL>
```

```
<Transition> ::= "if: " <Predicate> "then move to state " <string>
```

Predicate est un concept abstrait et n'a donc pas de syntaxe directe, et le concept *Predicate* dans Transition utilise donc les sous-classes de *Predicate*. De plus, la *string* utilisée après "*then move to state*" se doit de correspondre au nom d'un des états.

```
<AnalogPredicate> ::= "[analog predicate]" <string> <OPERATOR> "threshold " <int>
```

La *string* utilisée après "*[analog predicate]*" se doit de correspondre au nom d'un des AnalogSensor.

```
<DigitalPredicate> ::= "[digital predicate]" <string> " is " <SIGNAL>
```

La *string* utilisée après "*[digital predicate]*" se doit de correspondre au nom d'un des DigitalSensor.

```
<OPERATOR> ::= "equals" | "greater than" | "greater or equals" | "lower than" | "lower or equals"
<SIGNAL> ::= "high" | "low"
```

2.2.2 DSL interne

Voici une traduction "*BNF like*" de la syntaxe obtenue avec **Groovy** :

```
<App> ::= "bricks": <Brick>+ "states:" <State>+ "initial: " <State>
```

Brick est un concept abstrait et elle n'a pas de syntaxe directe. Le concept *Brick* dans App utilise les sous-classes de *Brick*. Le champ *initial* prend en paramètre un *State* qui désigne l'état initial et doit correspondre à un état précédemment créé.

```
<Actuator> ::= "actuator " <string> " pin " <int>
<AnalogSensor> ::= "analogsensor " <string> " pin " <int> " factor " <float>
<DigitalSensor> ::= "digitalsensor " <string> "pin " <int>
<State> ::= "state " <string> " means " <Action>+ <Transition>+
```

Action est un concept abstrait, il utilise donc les sous-classes d'*Action* : *AnalogAction* et *DigitalAction*. Néanmoins afin de donner un type à l'action en cours nous avons mis en place un système de vérification du type sous forme de condition qui est propre à la syntaxe d'une *Action*. Afin de garantir la possibilité d'ajouter plusieurs actions, nous avons utilisé les *"closure"* proposées par **Groovy**. Cette dernière est ici représenté par le bloc optionnel "[and X]" permettant de rappeler plusieurs fois une *Action*.

```
<Action>::= <string> " oftype " <string> "if (" <string> == 'analog' ")"
<AnalogAction>  [ else  <DigitalAction> ] [and <Action>]
   <AnalogAction> ::= " takes " <int>
<DigitalAction> ::= " becomes " <SIGNAL>

<Transition> ::= "from " <string> "to " <string> "when "
<Predicate> ::= <string> "oftype " <string> "if (" <type == "analog"> ")"
<AnaogPredicate>  [ else  <DigitalPredicate>  ] [and: <Predicate>]
```

Le concept *Predicate* dans *Transition* utilise les sous-classes de *Predicate* : *AnalogPredicate* et *DigitalPredicate*. Il se base sur le même principe de condition qu'une *Action* et a également été implémenté avec une *"closure"*. Le *<string>* utilisé après *"when"* doit correspondre au nom d'un *Sensor* existant.

```
   <AnalogPredicate> ::= "operator <OPERATOR> "threshold " <int>
<DigitalPredicate> ::= "is " <SIGNAL> "and " <Predicate><OPERATOR> ::= "==" | ">"
| ">=" | "<" | "<="
<SIGNAL> ::= "high" | "low"
```

3 Extension

3.1 Handling Analogical Bricks (**)

3.1.1 Analogical Sensors

Nous avons choisi d'implémenter l'extension *Handling Analogical Bricks (**)*. Ayant uniquement des briques digitales, nous avons dû adapter notre modèle en fonction. La différence majeure pour notre langage est l'ajout de valeurs numériques dans les comparaisons utilisées

pour les transitions et non uniquement de signaux. Ainsi comme nous pouvons le voir dans le diagramme (cf partie 2), nous avons modifié la classe `Predicate` en la rendant abstraite, pour avoir une classe fille `DigitalPredicate` pour les prédicats prenant des signaux digitaux ; et une classe fille `AnalogPredicate` pour les prédicats prenant des valeurs analogiques. Pour la partie comparaison, un opérateur est seulement nécessaire pour les prédicats analogiques. En effet, nous pouvons vérifier la valeur d'un signal digital avec l'opérateur d'égalité pour les `DigitalPredicate`.

Pour la lecture des valeurs analogiques, nous avons choisi d'utiliser un facteur multiplicateur pour être en mesure d'interpréter la valeur renvoyée par le capteur (en voltage). Nous avions tout d'abord pensé à nous servir directement d'une formule permettant de convertir le voltage en degrés. Mais celle-ci impliquait de recueillir la valeur du capteur préalablement puis de l'inclure dans la formule et enfin d'utiliser ce résultat comme valeur de comparaison au sein d'un prédicat d'une transition. Ce système impliquait un changement radical de fonctionnement de notre modèle et empêchait l'extension vers de nouveaux capteurs à moindre coût. Or nous avons pu trouver des exemples et de la documentation où les valeurs de ces capteurs sont associées à un facteur pour avoir un sens plus concret (par exemple une température en degrés Celsius ou un pourcentage de luminosité). Nous avons donc choisi d'intégrer le facteur de conversion à l'AnalogSensor.

Scénarios associés :
— Détection d'un changement de température (Scénario 5).
— Détection d'un changement de luminosité (Scénario 6).

3.1.2 Analogical Actuators

Dans l'objectif de gérer également des *actuators* analogiques, nous avons également ajouté de l'héritage sur les actions. Cela se justifie par le fait que pour un actuator digital nous avons simplement besoin de définir si le signal est `HIGH` ou `LOW` contrairement à une cible analogique où il est nécessaire d'utiliser une valeur numérique. Cette fonctionnalité permet par exemple de définir l'intensité d'une LED ou la note d'un buzzer. Nous avons donc pour cela rendu la classe `Action` abstraite et implémenté deux classes filles `DigitalAction` et `AnalogAction` permettant de répondre aux besoins cités précédemment.

Scénario associé :
— Envoie d'une valeur analogique à une LED (Scénario 7).

3.2 Prise de recul

Le premier point fort de notre conception est l'utilisation de l'**héritage**. En effet, cela permet de gérer aussi bien les matériels analogiques que digitaux (sensors, actions, prédicats). Cela a même permis dans le DSL interne l'utilisation de types génériques tels que `Predicate<T>` afin de s'assurer qu'un prédicat "digital" référence un détecteur digital (`DigitalPredicate extends Predicate<DigitalSensor>`). L'héritage nous permet par ailleurs d'être plus extensible. Les deux grands types de *sensors/actions* sont représentés mais nous pouvons tout de

même imaginer des prédicats plus complexes qui pourraient étendre ces concepts.

En ce sens, nous pensons que la gestion des transitions avec la notion de **prédicat** est très intéressante bien qu'elle fut difficile à développer. Cette dernière est représenté par un tuple `[sensor, operator, value]` qui permet de répondre aux besoins de tous nos scénarios (cf partie 4).

Pour revenir sur l'impossibilité d'implémenter de la généricité sur MPS, elle est relevée dans le poste suivant : ici directement par le support. Dans ce même poste, la personne auteur de la question a pu trouver un *work around* pour son cas d'utilisation en se servant de contraintes. Nous n'avons pas réussi à appliquer un tel work around pour notre langage. Pour pouvoir tout de même obtenir le résultat désiré, nous avons directement lié les DigitalPredicates aux DigitalSensors, et les AnalogPredicates aux AnalogSensors. Nous aurions préféré utiliser la classe mère pour rester cohérent avec le reste de notre diagramme et garder une conception plus extensible, qui justifie aussi l'héritage.

4 Scénarios

Nos DSLs permettent d'effectuer les 4 scénarios basiques suivants :

1. **Alarme simple :** lorsqu'on presse le bouton en le gardant enfoncé, on déclenche un buzzer et une LED, lorsqu'on relache ce bouton, le buzzer et la LED s'éteignent.
2. **L'alarme a double vérification :** le buzzer se déclanchera uniquement si les deux boutons sont pressés "en même temps". Si un des deux boutons est relaché, le buzzer s'éteint.
3. **Interrupteur :** le fait d'appuyer sur le bouton va allumer la LED qui s'éteindra lors du prochain appuit.
4. **Alarme multi-états :** appuyer sur le bouton 1 fois déclenche le buzzer. Appuyer encore une fois éteint le buzzer et allume la LED. Appuyer une troisième fois éteint la LED. Si on rappuie sur le bouton, cela redéclenchera le buzzer et ainsi de suite.

   Avec l'implémentation de l'extension *Handling Analogical Bricks*, nous avons ajouté 3 scénarios qui y sont liés :
5. **Capteur de luminosité :** un seuil de luminosité est indiqué au capteur, avec son facteur de traduction. Quand la luminosité franchit ce seuil, le buzzer se déclenche. Quand la luminosité repasse au delà du seuil, il s'éteint.
6. **Capteur de température :** un seuil de température est indiqué au capteur, avec son facteur de traduction. Quand la température franchit ce seuil, le buzzer se déclenche. Quand la température repasse au delà du seuil, il s'éteint.
7. **Interrupteur analogique :** puisque les éléments de sortie Arduino que nous utilisons (la LED et le buzzer) peuvent aussi être utilisés en tant que sorties analogiques, nous allumons la LED lorsque le potentiomètre dépasse un certain seuil. Cependant cette fois la LED est allumée en recevant une valeur analogique (et est donc branchée sur un port analogique). Elle s'éteint donc en recevant la valeur 0 quand le potentiomètre est tourné en dessous du seuil.

5 Analyse

5.1 Implémentation du DSL

Nous avons fait le choix de développer **deux kernels**, un pour chaque DSL. Cela a pour avantage de **travailler de manière totalement indépendante** dans l'équipe entre les deux types de DSL. Nous avons pu avancer chacun à notre rythme en agrémentant le kernel au fur et à mesure, sans bloquer une autre partie de l'équipe ou en la forçant à adapter son code. Le point négatif de cette décision est qu'il faut développer deux fois le même kernel qui aurait pu être partagé et donc potentiellement **moins de cohérence entre les deux domaines sémantiques**.

5.2 Technologies

5.2.1 JetBrains MPS

Le choix de notre *Language Workbench* s'est fait après avoir testé XText et JetBrains MPS. Ces derniers semblaient être les plus matures pour respectivement les langages non projectionnels et projectionnels. Or un tel workbench a pour but de nous fournir des services pour que le développement soit le plus simple pour nous. En ce sens, nous avons finalement décidé d'utiliser **MPS** car il fournissait de nombreux exemples et nous cadrait en étant projectionnel.
Après avoir choisi MPS, nous avons réussi à implémenter et comprendre les bases assez rapidement. Sa structure, avec les éléments appelés Concepts, Editors, Constraints et TextGen, facilitent la compréhension du procédé de développement. L'aspect projectionnel nous a forcé à être rigoureux à chaque étape, et nous a permis de toujours relier nos éléments en respectant le diagramme.
Cependant MPS a aussi des inconvénients que nous avons ressentis au cours de ce projet. Lors du début du développement ou la réalisation du POC, nous n'avions pas rencontré de problème majeur, donc nous n'avions pas réalisé à quel point la communauté est restreinte. En effet, nos recherches sur Google ont rarement été fructueuses. C'est donc le principal point négatif que nous avons rencontré en utilisant MPS. Cela a eu pour impact de ralentir le développement dès que nous avons dû dépasser les fonctionnalités basiques expliquées dans le tutoriel du cours ou ceux trouvés en ligne. Enfin, nous notons que pour des personnes habituées à programmer dans un langage comme Java, le fait que certaines fonctionnalités, telles que la généricité, n'existent pas dans MPS, nous force à penser différemment, ce qui n'est pas un mal en soi, mais apporte un handicap pour un développement plus avancé, dans le temps imparti.

5.2.2 Groovy

Nous avons choisi **Groovy** pour notre DSL interne après une étude de l'état de l'art. Notre deuxième pour se portait vers Scala mais Groovy était selon nous le meilleur compromis entre facilité de prise en main, documentation et limitations techniques. Bien qu'il puisse s'avérer difficile à comprendre, Groovy offre des mécanismes puissants tels que les *closures* pour déclarer n fois un prédicat par exemple. Il permet également un typage dynamique qui nous a été très utile face aux nombreux héritages mis en place et sur lesquels nous avions un besoin de contrôle

des types. Groovy possède aussi l'avantage d'être une surcouche de Java, inspiré du python mais également extrêmement ressemblant à la syntaxe Java, qui est le langage que nous connaissons le mieux au sein du groupe.

5.2.3 Prise de recul

Chaque type de DSL possède ses avantages et ses inconvénients. Il est plus dur se tromper avec un DSL externe, particulièrement avec un *language workbench* projectionnel, mais un DSL interne est plus facile à appréhender. Un DSL interne est limité par les possibilités de son langage hôte, notamment plus limité dans sa syntaxe, mais un DSL externe peut très vite devenir un *language cacophony*. Ainsi, notre jugement est purement subjectif et dépend du sujet que nous avons traité. Notre équipe est unanime sur la préférence du **DSL interne**. Les problèmes techniques sur JetBrains MPS ont été plus récurrents que pour Groovy. Cela est dû en partie au fait que nous sommes familiers avec les langages de programmation. Néanmoins, Martin Fowler dans son livre *Domain Specific Languages* nuance ces propos et dit que cette différence entre types de DSLs n'est pas si marquée pour les développeurs. Ainsi, il serait certainement plus intéressant d'utiliser un DSL externe pour un projet sur le long terme nous permettant d'absorber cette courbe d'apprentissage.

6 Travail d'équipe

Nous nous sommes répartis équitablement le travail et les différentes tâches entre les membres de l'équipe. Aussi, nous avons voulu que chaque membre du groupe travaille sur les deux types de DSL. Voici la répartition du travail au sein du groupe :

- Théo : tutoriel XText, développement des scénarios sur MPS avec Laura.
- Florian : tutoriel MPS, développement des scénarios en Groovy.
- Laura : étude de l'état de l'art DSLs internes, développement des scénarios sur MPS avec Théo et également de l'extension et des scénarios correspondants en Groovy.