DSL : SENSOR SIMULATION

Génovèse Matthieu Liechtensteger Michael Chennouf Mohamed

February 2018

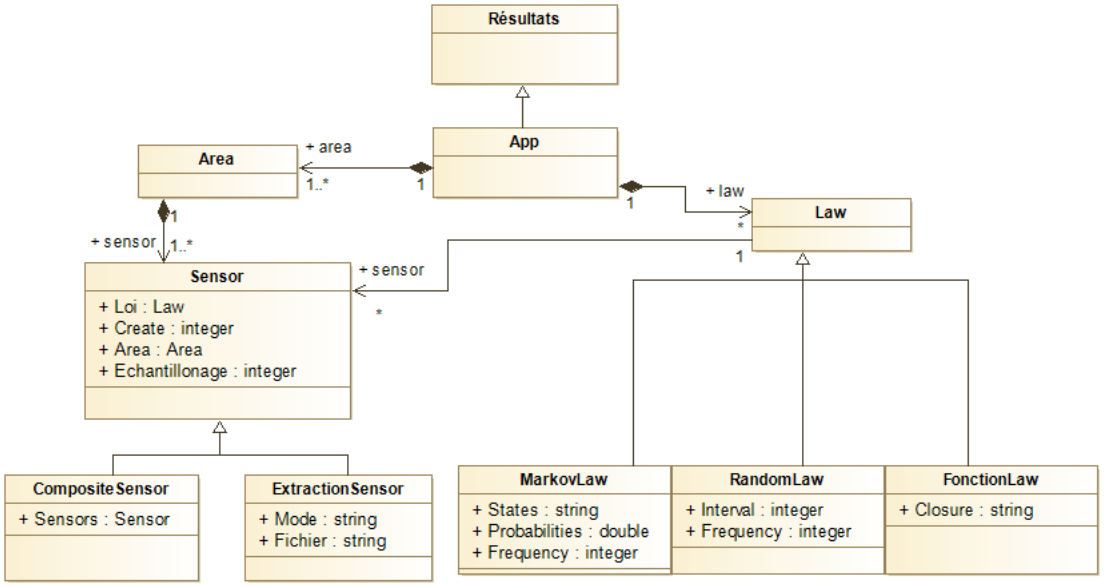
1 Introduction

Le but de ce projet est de réaliser un DSL (Domain Specific Language) dans le contexte d'une simulation de capteurs. Notre language doit permettre de définir plusieurs capteur ainsi que plusieurs lois servant à générer les valeurs des capteurs. La simulation doit avoir une certaine durée dans le temps et chacune des valeurs obtenues pour les capteurs à un instant donné doit être enregistrée dans une base de données InfluxDB, ces données peuvent par la suite être visualisées sur Grafana.

2 Description du langage

2.1 Diagramme de classe

Voici le diagramme de classe de notre langage :



2.2 Syntaxe du langage

Nous allons ici décrire tous les éléments que l'on peut déclarer dans notre langage et leur syntaxe dans notre DSL.

2.2.1 Déclarer des lois

Notre langage permet de déclarer des lois que l'on va pouvoir associer à nos capteurs, qui vont définir leur comportement. Toutes les lois possèdent un nom et différents attributs propres à chaque loi. Pour définir une loi, nous utilisons le mots clé "type de la loi" + "law". Les types de lois prenant des paramètres différents, nous avons préféré créer des objets différents propres à chaque loi (qui partagent cependant une interface commune) pour éviter de devoir gérer un objet loi trop complexe, qui pourrait représenter tous les types de lois. Il existe plusieurs types de lois définissables par l'utilisateur :

Loi de markov

La loi de markov possède différents états que peuvent prendre le capteur. Pour changer d'état, on définit dans la loi des probabilités qui permettent au capteur de changer d'état. La loi possède également une fréquence, permettant de définir à quel période de temps les capteurs effectuent une transition d'un état à un autre. Les états peuvent être des chaînes de caractères ou des nombres.

Exemple de loi de markov :

```
markovLaw "markovLaw" states (["ensoleillé","nuageux","orageux"]) transi
([[0.1,0.2,0.7],[0.3,0.5,0.2],[0.4,0.5,0.1]]) frequency 1 by min
```

Ici, nous créeons une loi de markov appelée markovlaw, qui possède trois états que sont ensoleillé, nuageux ou orageux. Pour changer d'état, une matrice de probabilité est définie par l'utilisateur après le mot clé "transi". Chaque ligne de cette matrice correspond à un état de la loi, et représente les différentes probabilités pour cet état de rester dans l'état actuel ou changer d'état.

Loi random

La loi random possède un interval d'entiers. L'état du capteur prend une valeur aléatoire comprise dans l'interval défini dans la loi. Cette loi possède également une fréquence permettant de décider à quel interval de temps la valeur du capteur doit être modifiée.

Exemple de loi random :

```
randomLaw "randomLaw" interval ([1,10]) frequency 1 by min
```

Dans cet exemple, les capteurs associés à cette loi pourront prendre comme valeur un entier compris entre 1 et 10, et la valeur changera toutes les minutes.

Loi fonctionnelle

La loi fonctionnelle permet de définir la valeur du capteur en fonction du temps de la simulation. Contrairement aux autres lois, celle-ci ne possède pas de fréquence. L'utilisateur peut définir des instructions qui vont représenter le comportement du capteur en fonction de conditions sur le temps de la simulation. Pour gérer l'écriture de différents conditions par l'utilisateur, nous utilisons un type d'objet appelé "closure" définissable en Groovy.

Exemple de loi fonctionnelle :

```
functionLaw "fonctionLaw", {
    t ->
            if (t > 10 && t < 30) return 1
            if (t < 10) return 17
            if (t < 300 && t > 30) return 72
            if (t > 300) return 99
}
```

Ici, la variable t représente le nombre de pas de la simulation. A chaque pas, la loi vérifie quelle condition sur le temps est respectée, et associe la valeur du return au capteur. Si plusueurs conditions sont respectées en même temps, la condition définie en premier sera choisie.

2.2.2 Déclarer des capteurs

Il existe 3 types de sensors différents :
-Les sensors "normaux" ils tireront leurs comportement de la loi qui leur est associée.
-Les sensors "extractions" ces sensors là n'ont pas de loi à proprement parlé, ils génèrent leurs valeurs en fonction d'un input (dans le cadre de notre projet c'est soit un csv, soit un json).
-Un 3ème type de sensor existe également, nous en reparlerons en détail dans la section "Fonctionnalité additionnelle".

Sensor classique

Voici un exemple de sensor "normal" :

```
sensor "temps" law "markovLaw" create 10 area 1 echantillonage 1 by s
```

Ici nous déclarons directement un groupe de 10 sensors nommés de "temps0" à "temps9". Ces sensors vont générer leurs valeurs en fonction de la loi "markovLaw" définit plus haut dans la section "Déclarer les lois". Le mot clé "area" sert a associé le sensor à une zone, on peut définir plusieurs groupe de sensors différents associés à la même zone. Pour finir nous attribuons un échantillonage à notre sensor qui est le nombre d'échantillon par unité de temps que va envoyer le sensor. Plusieurs unités de temps ont été implémentées, en voici la liste :
- x by ms (x par milliseconde)
- x by s (x par seconde)
- x by min (x par minute)
- x by h (x par heure)
- x by d (x par jour)

Sensor Extraction

Nous allons maintenant voir comment déclarer un "extractionSensor" pour cela nous allons utiliser le fichier "data.csv" qui contient :

```
extraction1,0,1
extraction2,0,2
extraction3,0,3
extraction1,1,11
extraction2,1,22
extraction3,1,33
extraction1,2,111
extraction2,2,222
extraction3,2,333
extraction1,3,1111
extraction1,4,11111
extraction1,6,145
extraction1,5,4566
```

le format de fichier utilisé est de la forme "nom, temps, valeur". Voici comment on utilise un tel fichier dans notre DSL :

```
extractionSensor "extracteur1" mode "csv" path "E:\\data.csv" sensor "extraction2"
create 1 area 1 timeunit s
```

Ici nous déclarons un "extractionSensor" avec pour nom "extracteur1", on spécifie que le mode de lecture du fichier est "csv" (il existe également un mode "json") on renseigne ensuite le chemin du fichier qui contient les données puis comme pour les sensors "normaux" on indique un nombre de sensors à créer et une area. Il faut également spécifier un paramètre "timeunit" (ms, s, min, h, d) qui va correspondre à l'unité de temps stocké dans le fichier de donnée, ainsi si l'utilisateur à des données en millisecondes, il n'a pas besoin de changer son fichier mais uniquement de spécifier "timeunit ms", le simulateur se chargera de faire les conversions nécessaires en backend lors de la simulation. Dans cette exemple, le sensor "extracteur1" prendra uniquement les valeurs et les temps des lignes correspondant au nom "extraction2".

Voilà un exemple de fichier json pour notre DSL :

```
{
    "sensors": [
        {
          "name": "json1" ,
            "values":[
                {"time":0 ,"value":1},
                {"time":1000 ,"value":11},
                {"time":2000 ,"value":111}
                {"time":3000 ,"value":1111}
                {"time":7000, "value": 154556}
            ]
        },
        {
             "name": "json2",
             "values":[
                 {"time":0 ,"value":2},
                 {"time":1 ,"value":22},
                 {"time":2 ,"value":222}
                 {"time":3 ,"value":2222}
             ]
        }
    ]
}
```

Il est également possible de rajouter du bruit dans la lecture d'un fichier avec la commande :

```
myExtractionSensor.addNoise([0,10])
```

Avec myExtractionSensor un extractionSensor déjà défini. Cela va ajouter un bruit allant de 0 à 10 pour chaque valeur de la simulation. Il est également possible de définir une zone à récupérer avec la commande suivante :

```
myExtractionSensor.addMinOffset(5)
myExtractionSensor.addMaxOffset(10)
```

Dnas ce cas, les valeurs qui seront enregistrés dans la base de données sont ceux ayant un temps compris entre 5 et 10. Il est également possible de ne préciser que l'offset minimum ou l'offset maximum.

2.2.3 Lancer la simulation

Pour lancer une simulation, l'utilisateur donne une date de début et une date de fin, voici la commande :

```
runApp "17/02/2018 12:10:00 PM" to "17/02/2018 12:10:43 PM"
```

Le format de la date est important, seul celui de cet exemple sera accepté. Cette commande a pour effet de convertir en timestamp la date de debut et la date de fin. Avant le lancement de la simulation, tous les temps des sensors sont initialisés avec le timestamp de la date de debut. A chaque tour de simulation, le temps de chacun des sensor est incrémenté de leurs échantillonages respectifs. Une fois qu'un sensor à atteint le timestamp de la date de fin, il entre dans un état "terminé" et il arrête de générer des valeurs. La simulation se termine une fois que tous les sensors sont dans l'état "terminé"

2.2.4 Fonctionnalité additionnelle

Pour la fonctionnalité additionnelle, nous avons choisi le Composite Sensor. Cet ajout permet à l'utilisateur de créer un nouveau type de capteurs, appelés capteurs composites. Ces capteurs fonctionnent différement des autres capteurs, car leurs valeurs dépendent d'autres capteurs. Pour définir un capteur composite, l'utilisateur doit préciser un ensemble de capteurs qui va être associé au capteur composite. L'utilisateur précise ensuite une fonction qui est appliqué sur le groupe de capteurs, comme la somme ou la moyenne.

Exemple de capteur composite :

```
compositeSensor "myCompositeSensor" sensor "markovsensor" function "average" create 1 area 1 echantillo
```

Ici, on déclare un sensor composite qui possède un groupe de sensors appelé "markovsensor" et une function appelée "average". Cela signifie que la valeur du sensor composite sera égale à la moyenne de tous les sensors associés au sensor composite. Le sensor composite possède comme les autres sensors un nombre représentant combien de sensors composite seront créés, un lieu et un échantillonage. Il est également possible de mettre à jour les sensors associés au composite sensor avec la commande :

```
myCompositeSensor.addSensors("randomSensor")
```

Avec randomSensor un groupe de sensors déjà défini.

2.3 Gestion d'erreurs

Dans le développement de notre DSL, nous avons trouvé important d'offrir une bonne gestion des erreurs éventuelles de l'utilisateur. Nous avons capturé le plus d'erreurs possible. Chacune de ces erreurs est detectée à la lecture du programme, si l'utilisateur à fait une erreur la simulation ne démarrera pas. Voici un exemple d'un fichier qui va générer des erreurs :

```
markovLaw "markovLaw" states ([10,-158,180])
transi ([[0.3,0.2,0.7],["totovaalapeche",0.5,0.2],[0.4,0.5,0.1]]) frequency "f" by "er"
randomLaw "randomLaw" interval ([1,50]) frequency 1 by s
functionLaw "fonctionLaw", {
    x ->
        if (x > 1 && x < 3) return 1
}
sensor "markov" law "markovLaw2" create "ff" area 1 echantillonage "fd" by s
sensor "fonctionel" law "fonctionLaw" create 1 area 1 echantillonage 1 by s
sensor "random2" law "randomLaw" create 3 area 1 echantillonage 1 by s
extractionSensor "toto2" mode "csv54" path "554" sensor "toto" create 1 area 1 timeunit "5545"
compositeSensor "compositeSensor" sensor "random2" function "average" create 1 area 1 echantillonage 1
compositeSensor.addSensor("mar")
runApp "17/02/2018 12:10:54 PM" to "17/02/2018 12:10:43 PM"
```

L'output chez l'utilisateur sera le suivant :

```
-----ERREURS DE COMPILATION-----

Le paramètre f est invalide! L'unité de temps er n'existe pas !(ligne 1)
la somme des probabilités de [0.3, 0.2, 0.7] ne fait pas 1 !
Le paramètre totovaalapeche est invalide! Le paramètre ff est invalide!
la datalaw null n'existe pas !(ligne 7)
mode de lecture inconnue ! (les modes acceptés sont json et csv)(ligne 10)
Le fichier spécifié est introuvable ! L'unité de temps 5545 n'existe pas !(ligne 10)
Le sensor mar n'existe pas !(ligne 12)
La date de début doit être supérieur à la date de fin !(ligne 13)
```

Pour effectuer ces vérifications, nous avons une classe Java (ErrorDetection) qui possède tout un tas de méthode de verification. Par exemple pour la déclaration d'une loi de markov, avant de créer l'objet MarkovLaw réellement nous allons lancer cette méthode :
this.erreurHandler.checkMarkovImplementation(states, map, f, unit)
où states est la liste d'états, map la liste de probabilités, f la fréquence et unit l'unité de temps.

3 Analyse critique

3.1 Pourquoi groovy ?

Pour ce projet, nous avons choisi un DSL interne et avons utilisé le langage groovy. Groovy nous a semblé être un bon choix car l'utilisation d'un baseScript associé à un binding facilite fortement la mise en place d'un DSL. Celà nous a permis de déclarer des variables déjà assignées à la création du programme de l'utilisateur (notamment pour les unités de temps acceptées), mais celà nous a également permis d'autoriser des syntaxes de ce type : sensorComposite.addSensors("sensorname"). Lorsque le binding voit ce type de syntaxe, il cherche dans ses variables si une variable de type "SensorComposite" qui a pour nom "sensorComposite" existe, et si c'est le cas il appelle directement la méthode addSensors("sensorname"). Un tel comportement nous permet d'autoriser certaines méthodes et d'en interdire d'autres selon les conditions que l'on souhaite ce qui facilite grandement l'implémentation d'un DSL. Enfin le dernier avantage de groovy selon nous est l'utilisation des closures, notamment pour les lois fonctionnelles. La syntaxe que nous avons présentée précédement dans le rapport au sujet des lois fonctionnelles est tout simplement une closure qui sera appellée à chaque tour de simulation avec la valeur de temps correspondante, cela nous évite de mettre au point un parseur de formules mathématiques et cela nous permet d'utiliser toutes les primitives de Groovy (cosinus, sinus, valeurs absolue etc...)

3.2 Choix d'implémentation

Dans notre implémentation du Sensor Simulation Language, nous avons décidé de différencier la déclaration des capteurs avec leur comportement. Cela permet de définir plusieurs groupes de capteurs avec potentiellement le même comportement. Nous avons choisi de fonctionner en groupes de capteurs, car nous voulions pouvoir facilement simuler plusieurs capteurs dans une même zone ayant un comportement spécifique. Nous avons également pris la décision de différencier certains types de capteurs pour plusieurs raisons :
- leurs arguments étaient totalement différents, donc permet de rendre le code en Groovy plus facilement lisible.
- l'utilisateur n'a pas de doute concernant le rôle de chaque type de capteur.

Concernant la gestion du temps au niveau des lois et des capteurs, les fréquences s'expriment en unité de temps, de la forme "nombre entier" by "unité de temps". Cela permet à l'utilisateur de définir un temps dans une syntaxe compréhensible et qui reste cohérente par rapport aux temps qu'il doit donner pour exécuter la simulation.

4 Discussion du DSL dans le cadre SSL

Nous pensons que l'utilisation d'un DSL dans le cadre de la simulation de sensor est plutôt utile, car cela permet à un utilisateur non initié a la programmation mais qui à quand même des connaissances dans le domaine des Sensors et de la simulation (il saura ce qu'est une chaine de markov, un échantillonage, une fréquence etc...) de pouvoir facilement paramétrer une simulation cohérante. De plus, l'utilisation de Groovy permet de définir une grammaire facile d'utilisation pour l'utilisateur, sans les inconvénients des DSL externes (notamment le fait de ne pas avoir à développer de parseur pour une grammaire)