Rapport DSL

Sensor Simulation Language (SSL)

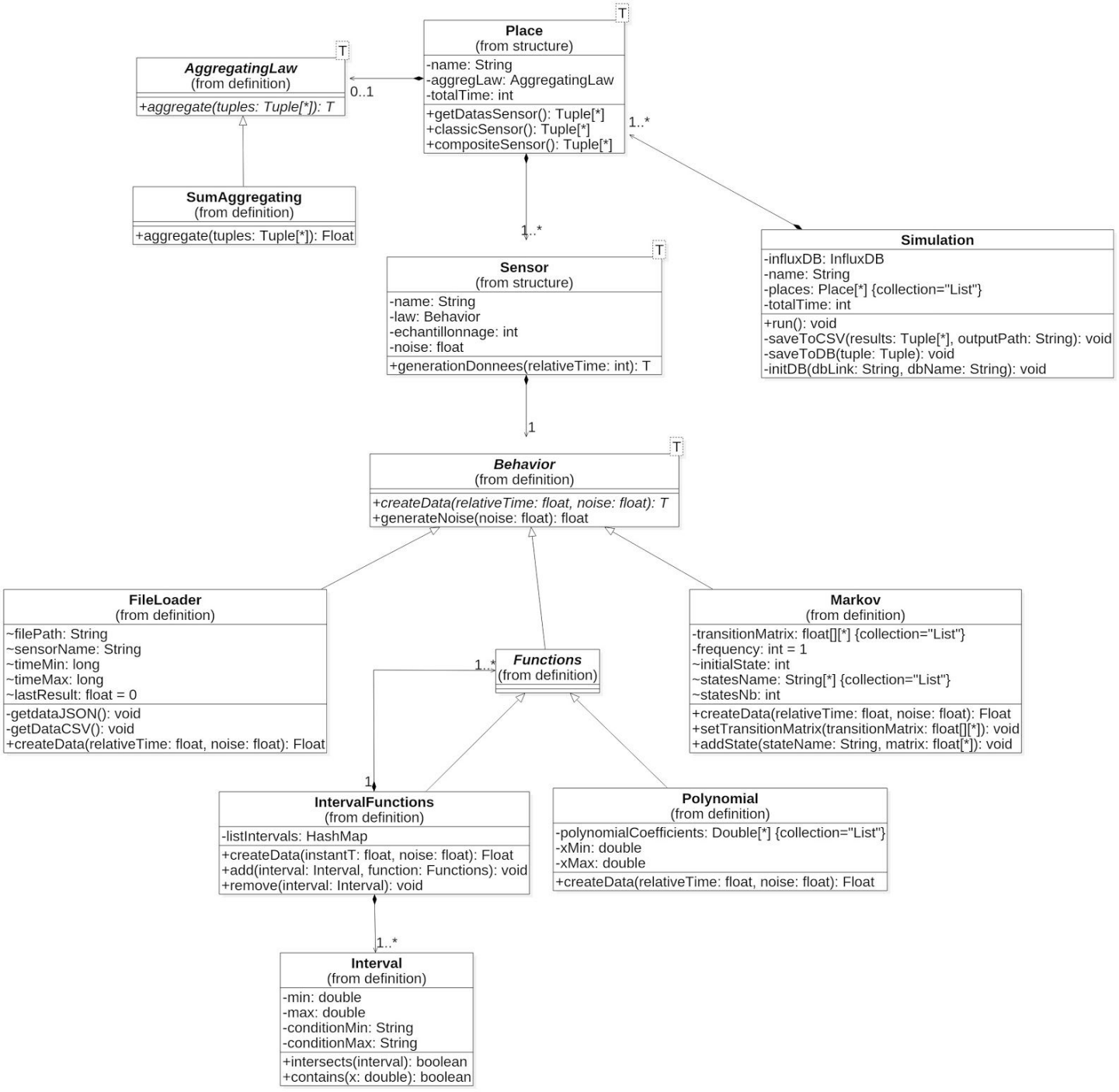
Maxime Dejeux
Anthonny Giroud
Khadim GNING

1. Introduction

Les objets connectés sont de plus en plus présents dans notre vie de tous les jours et avec eux de nombreuses applications et mêmes écosystème se développent (maison connectées, bâtiment intelligent, Smart cities pour ne citer qu'eux). Il va s'en dire que les données générées par ce type d'objet vont en grandissant. Développé des solutions (informatique) dédiées aux réseaux d'objets connectés n'est pas une tâche facile, mais tester ces solutions, est une tâche encore plus complexe.
Le DSL que nous avons créé a pour objectif de permettre de simuler des réseaux de capteurs. Le but étant de permettre aux entreprises développant de telles solutions de tester leur produit sans avoir à récolter les données de réel capteur avant ça. Notre solution offre bien évidemment la possibilité de rejouer des sets de données si le client en possède déjà, mais aussi d'en créer de nouveaux en définissant des lois (chaînes de Markov, polynôme ou autre) qu'on associe à des capteurs. Les capteurs sont regroupés par lieu afin de se rapprocher des cas d'utilisation auxquels nos clients seront confrontés ("sensing infrastructure").
Pour développer notre DSL nous avons utilisé les langages Groovy et Java. Le projet est séparé en deux modules, un module Kernel qui gère tout ce qui a trait à la simulation, la génération de donnée et un module SensorSimDSL qui permet à l'utilisateur de définir, en utilisant un langage plus ou moins naturel les paramètres de la simulation à lancer.
Dans la suite de ce rapport, nous allons expliciter le fonctionnement de ces différents modules et justifier nos choix d'architecture et de développement.

2. Model du domaine

Le modèle est la description du système et de ces fonctionnalités. Il a donc pour but de faciliter la compréhension du système.
Pour des raisons de lisibilité nous avons décidé de ne pas afficher les accesseurs et mutateurs dans chacune des classes du diagramme de classes ci-dessous.



2.1. Définition de la simulation

Nous avons choisi de définir une simulation comme étant un ensemble de lieux, chaque lieu est défini comme étant un ensemble de capteurs. Chaque capteur intègre une loi qui peut donc être différentes des autres lois des capteurs contenus dans le même lieu, cela nous permet de pouvoir créer des lieux plus complexes et plus réalistes. Nous pouvons donc définir par exemple un bâtiment modélisant un ensemble de classe contenant des capteurs

d'occupations (chaîne de Markov) et des capteurs de température (qui peuvent être modélisé avec des fonctions polynomiales).
La classe “Simulation” contient une méthode “run” qui nous permet de créer les données de tous les capteurs de tous les bâtiments. Nous avons créé des capteurs de telle sorte que les utilisateurs peuvent définir un taux d'échantillonnage permettant de ne calculer, par exemple, qu’une nouvelle valeur toute les 5 secondes. Lorsque le taux d’échantillonnage ne permet pas de créer une nouvelle valeur alors c’est la dernière valeur créée qui est retournée.
A chaque seconde la simulation va récupérer les données de tous les capteurs, cela nous a permis d’implémenter l’extension plus facilement.

2.2. Création des données

Pour faciliter la création de données dans les capteurs par plusieurs lois possibles nous avons décidé de faire hériter toutes les lois de la classe “Behavior”. Cela nous permet d'implémenter dans chacune des lois la méthode “createData” permettant de générer les données et cette méthode sera appelée dans la méthode “generationDonnees” du capteur.
Etant donné que les lois et les options de “replay” ne retournent pas forcément une donnée du même type, nous avons donc décidé de mettre le type de retour de la méthode “createData” comme étant générique. Cela nous permet donc de pouvoir typer la donnée retournée par la méthode directement dans la loi.
Les capteurs peuvent pour chaque donnée générée ajouter un bruit dessus. Seuls les bruits numériques ont été implémentés pour le moment.

2.2.1. Rejouer les mesures existantes

Nous avons créé la classe “FileLoader” qui permet de pouvoir récupérer des données en provenance d’un fichier CSV ou JSON. De plus on peut sélectionner les valeurs dans le fichier entre un temps min et max, cela nous permet donc de pouvoir rejouer qu’une partie des données.

2.2.2. Loi des capteurs

2.2.2.1. Fonctions de modélisation

Nous avons défini une fonction de modélisation comme étant une liste d'intervalles et chaque intervalle ont une fonction associée. Nous avons choisi de créer pour chaque nouvelle fonction mathématique une nouvelle classe, même si ça restreint l’utilisateur sur la possibilité de créer toute sorte de fonction grâce à Groovy cela nous permet de gérer plus facilement les erreurs.

2.2.2.2. Modélisation des chaînes de Markov

Notre classe Markov a un tableau de float nommé “transitionMatrix” qui contient les probabilités d’un état à aller dans un autre état. Si le temps écoulé de la simulation modulo la fréquence d’autorisation de changement d’état est égal à zéro alors la méthode “createData” accorde le calcul d’un nouvel état, sinon l’ancien état est retourné.

2.3. Extension

Nous avons choisi l'extension "capteur composite" qui consiste à effectuer l'agrégation de valeur provenant de plusieurs capteurs. Le but de cette extension est d'agréger les données de plusieurs capteurs donc nous aurions pu utiliser le patron de conception composite pour créer un nouveau capteur composé de plusieurs capteurs, cependant dans la logique les capteurs agrégés proviennent de la même zone (comme l'exemple du parking) donc nous pouvons, en ajoutant une loi d'agrégation, assimilé une zone comme étant un capteur composite. Comme pour les lois des capteurs, les lois d'agrégation sont toutes héritées de la classe abstraite "AggregationLaw" et la logique d'agrégation se trouve dans les classes héritées de celle-ci.

3. Syntaxe concrète (Groovy)

Créer une simulation et définir sa durée :
```
simulation "simulation1" lasting 30
```

Créer un ou plusieurs groupes de capteurs :
```
instantiate "buildingA" and "buildingB"
```

Définir une loi suivant la fonction (-1/10)x²+0x+10 sur l'interval [-10,10] :
```
law "function3" function "Polynomial" between -10 and 10 with 10
and 0 and -1/10
```

Définir une loi à partir des données du sensor0 défini dans un fichier csv sur l'interval [0,10] :
```
law "function_csv" function "csv" from "../resources/resultat.csv"
with "sensor0" between 0 and 10
```

Définir une loi à partir utilisant plusieurs fonctions sur des intervals différents :
```
law "function_interval" function "Interval" follows function1 from
0 to 40 and "function2" from 41 to 60 and function3 from 61 to 100
```

Définir une loi de markov :
```
law "function_markov" function "markov" frequency 5 with "Sunny"
parameters ([0.1,0.9]) and "Rainy" parameters ([0.5,0.5])
```

Définir un capteur avec sa loi et sa fréquence d'échantillonage :
```
sensor "sensor2" follows "function2" every 1
```

Ajouter un ou plusieurs capteurs à un groupe :
```
add 1 sensors "sensor2" to "buildingA"
```

Lancer une simulation :
```
launch "simulation1"
```

Ci-dessus, on peut voir un exemple d'utilisation de la syntaxe concrète pour définir une simulation et l'exécuter. Pour notre syntaxe concrète nous avons choisi le langage Groovy, l'exemple montré est d'ailleurs un script Groovy. Librement inspiré, de la structure du DSL interne (Groovy) que nous avions étudié pour le kickoff lab, notre dsl repose sur la même

architecture. En terme de conception, nous avons mis plus de temps à mettre en place les bases du DSL qu'à développer les fonctions du Kernel qu'il appelle. Il a donc été conçu à partir des fonctionnalités existantes dans le Kernel.

Au final, en terme de structure on a un fichier SensorSimDSL qui fait le lien entre le script, notre modèle (ensemble des mots du langage) et le binding qui gère les variables de notre simulation. Pour être plus précis, SensorSimBinding nous permet de mapper les variables attachées au script pour partager des données entre le script et la classe appelante. En l'occurence, la classe appelante sera SensorSimBasecript. Le fichier SensorSimDSL permet également de définir des configurations de compilation via la classe SecureASTCustomizer. On peut par exemple autoriser ou interdire certains types d'opération dans le script (multiplication, division, …) ou encore des types de variables tout simplement (Array, String, …). Tout ceci est fait via un système de whitelist dans notre cas (une blacklist est également possible).

4. Analyse critique

Nous avons essayé d'obtenir un langage facile d'utilisation permettant de configurer des simulations avec précision. Nous avons choisi des mots clés cohérents avec leur utilisation afin de faciliter l'apprentissage de ce nouveau langage. Finalement, avec une vingtaine de mots l'utilisateur a les moyens de définir pléthore de simulation.

Nous sommes conscients que notre DSL présente, cependant, des faiblesses comme par exemple la restriction à n'utiliser que les types de fonction définies. De plus, l'implémentation de notre syntaxe nécessite sans doute un refactor, mais cette restriction met en péril la répartition du travail au niveau de la syntaxe concrète. Les fonctions liées à la définition d'une nouvelle loi ont des responsabilités (vérifications, comportements conditionnels lourd) bien trop importante par rapport au reste.
On a, actuellement, définis six types de fonctions. Grâce à la fonction intervalle, qui permet de créer une fonction composée de plusieurs autres fonctions, cette limite se ressent peu. Toutefois, c'est une amélioration que nous avons envisagée. En effet, l'intérêt d'utiliser le Groovy, outre le fait que c'est un langage embarqué tournant sur la JVM avec laquelle nous sommes habitués à travailler était de prendre des scripts interprétables en entrée. Ces scripts nous offrent la possibilité de laisser l'utilisateur définir ses propres fonctions en utilisant les closures. Cependant, cette approche n'était pas notre premier choix et nécessitait un travail important de vérification pour forcer l'utilisateur à saisir des fonctions censés par rapport aux simulations que nous devions créer.

L'approche choisie est donc un parti pris qui nous permet de gérer les différentes erreurs liés à la configuration de simulation donnée par l'utilisateur. De plus, notre gestion d'erreur bien que naïve, nous permet actuellement d'ignorer les erreurs qui ne mettent pas en péril l'exécution de la simulation et de donner un retour précis à l'utilisateur sur la source de son erreur. Nous sommes persuadés que cela aurait été bien plus difficiles à réaliser avec l'utilisation de closure.

Un dernier point, non négligeable, est que nous avons essayé de développer un langage facile d'utilisation pour l'utilisateur comme énoncé en début de partis. Rappelons que nos clients sont des développeurs de système gérant de large réseau de capteurs, l'enjeu pour eux est de pouvoir tester leur système de manière simple et rapide. Limiter leur choix d'action permet de rendre le DSL plus facile d'utilisation et dans certains cas (DSL plus large) plus efficace. En outre, nous sommes confiants du modèle d'objet choisis pour le DSL, ce dernier nous permet d'ajouter des fonctions facilement. Ces fonctions sont tout aussi facilement ajoutables à la syntaxe concrète ensuite.

5. DSL dans le contexte de langage de simulation de capteurs

Un DSL me paraît parfaitement approprié pour définir un langage de simulation de capteurs. En effet, les DSL sont conçues pour répondre aux contraintes d'un domaine d'application précis et quoi de plus précis que la simulation de capteurs.
En outre, les DSL apporte une couche d'abstraction permettant de gagner du temps, d'écrire des programmes plus courts et donc avec une sémantique plus accessible. Ils peuvent ainsi être utilisé par des personnes dont le métier n'est pas de développer du "software". Il existe d'ailleurs des langages dédiés aux domaines des mathématiques, de la médecine, de la cuisine et bien plus encore.

Le contexte de simulation de capteurs se prête bien à l'utilisation de DSL, car il y a beaucoup de paramètres configurables pour une simulation. Le DSL permet de personnaliser simplement et efficacement sa propre simulation. Il sert d'interface entre notre Kernel et l'utilisateur. On peut alors se demander s'il est plus intéressant de développer une interface graphique ou un DSL. Au niveau du coût de développement et de maintenance, nous pensons que le DSL est plus rapide à mettre en place. Par ailleurs, si on s'en tient strictement au sujet imposé, sur un même laps de temps, nous estimons qu'il eusse sans doutes été plus compliqué de développer une solution aussi extensible/flexible sous forme d'interface graphique.

D'un point de vue utilisateur, cependant, une interface graphique permettant d'ajouter des capteurs via un système de drag and drop sur une carte est peut-être plus parlant qu'un fichier de configuration tel que notre script groovy.

6. Répartition du travail

Anthonny : gestion de la base de données InfluxDB (sauvegarde des données) et Grafana.
Khadim : création de la syntaxe concrète (Groovy) et gestion des erreurs.
Maxime : création de la syntaxe abstraite "Kernel" (model du domaine).