Rapport Final - Team E

Danial BIN AHMAD - Rami AJROUD - Yasin EROGLU

Introduction

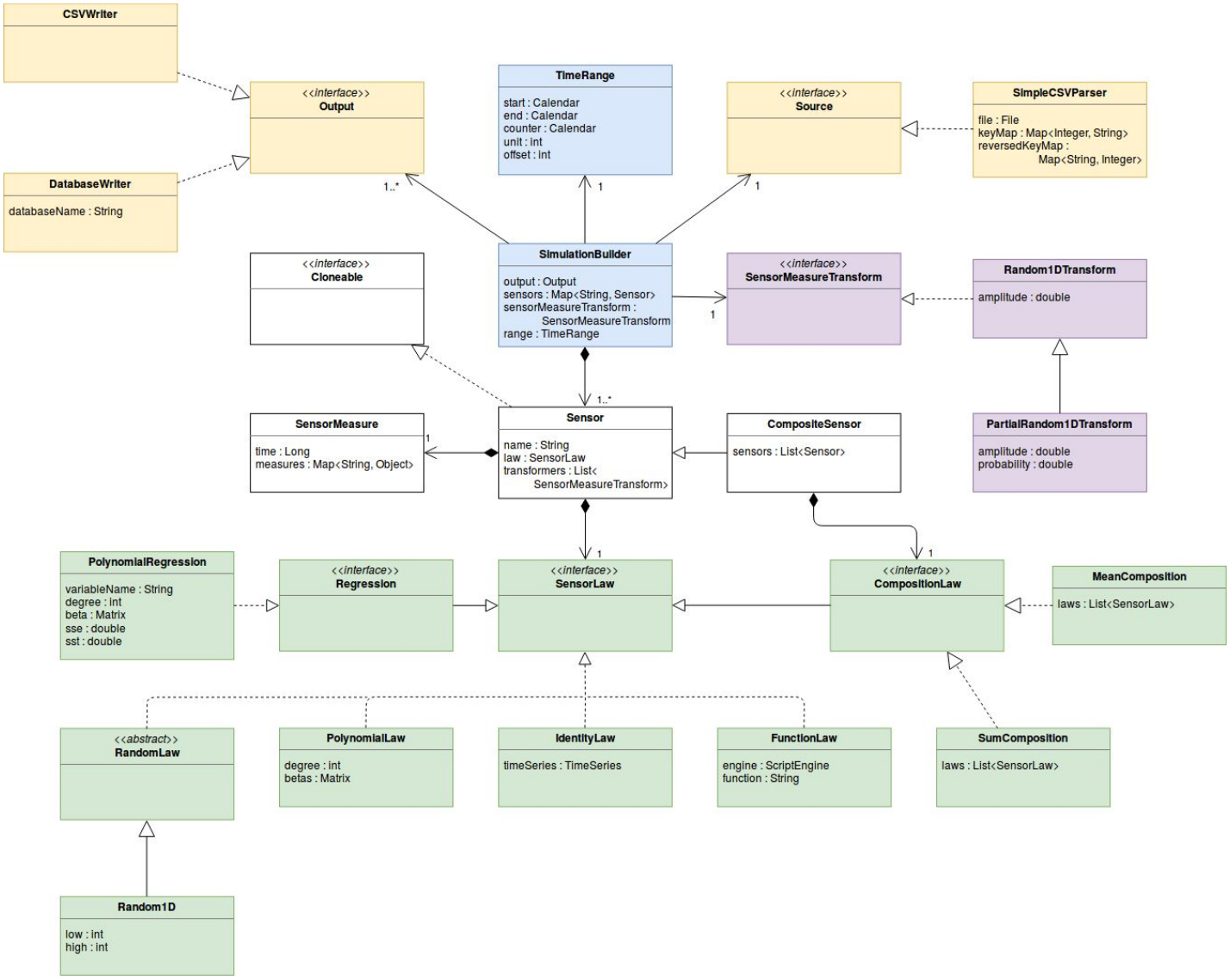
L'utilisation des appareils IoT pour construire la ville de future, ville intelligente, devient une tendance très populaire de nos jours. La mise en œuvre massive de ces appareils a engendré de nouveaux défis et exacerber ceux existants, parmi lesquels, ayant un intergiciel qui est non seulement capable de gérer les données générées, mais aussi capable de simuler des données du monde réel. Inspiré par ce problématique, le but de notre projet est de pouvoir répondre à ce besoin en utilisant l'approche DSL.

Ce document présente le travail produit par notre groupe dans le cadre de la matière Langage Dédié (*Domain Specific Language*, DSL). Nous montrons successivement le diagramme de classe sur lequel notre langage est basé, puis la syntaxe implémenté, l'extension mis en place, l'analyse critique de notre choix de technologie, ensuite la conclusion du projet.

Diagramme de classe

Nous avons ici représenté en couleur les différents éléments composant notre système :

- En orange, nous avons la gestion des entrées et sorties de notre simulateur.
- En bleu, la gestion du simulateur elle-même.
- En violet, nous avons la gestion des transformations applicables sur les mesures des sensors (ici un bruit aléatoire appliqué à toutes les mesure ou partiellement).
- En blanc, nous avons notre représentation des sensors dans notre système et des mesures qu'ils fournissent, une mesure étant caractérisée par un instant t et un ensemble de données labellisées.
- Enfin, en vert nous avons notre description des lois qui régissent les capteurs, un capteur n'utilisant qu'une loi (identité pour le replay, *PolynomialLaw* pour un capteur décrit par un polynôme, etc).



Notre implémentation consiste en l'instanciation des capteurs (*Sensor*) qui vont contenir chacun une loi (*SensorLaw*) définissant leur comportement et nous permet de récupérer des mesures (*SensorMeasure*). Chaque capteur possède des transformations (*SensorMeasureTransform*) qui seront appliquées sur les mesures. Ces mesures seront concaténé dans une série (*SensorMeasureTransform*) temporelle (*TimeSerie*) qui sera écrite dans une des sorties (*Output*).

L'objet *SimulationBuilder* sera le chef d'orchestre permettant de définir les paramètre de simulation et de la lancer.

Ces objets sont définis dans un kernel Java et sont lié à un module groovy qui est spécialisé dans la syntaxe du langage.

Un développeur pourra, s'il le souhaite, créer (en java ou en groovy) une extension au langage et l'intégrer dans la mesure où les fonctionnalitées qu'il désire ajouter implémentent les interfaces concernées. Par exemple: si on veut créer une nouvelle loi, alors il faudra implémenter *SensorLaw*.

La syntaxe

Nous avions initialement développé une première syntaxe pour notre DSL basée sur la technique de méthod chaining de Groovy. Cette technique consistant en l'enchaînement de mots clés nous à permis de définir des paramètres de simulation et de capteurs de manière assez naturelle. Cependant, nous avons rencontrés des difficultés lors d'ajout de nouvelles fonctionnalitées (bruit, composition). La première difficulté est l'impossibilité d'ajout d'extension externe à notre code. De plus, cette approche par enchaînement de mots clé nous limitait dans l'utilisation de fonctionnalité plus complexe que la simple déclaration de paramètres. Enfin, après inspection par des pairs, nous avons remarqué que la syntaxe n'était pas suffisamment expressive.

Nous avons donc perfectionné notre syntaxe pour obtenir la syntaxe suivante.

La première instruction pour toute simulation est la déclaration des paramètres de celles ci qui sont son nom, l'interval de temps sur laquelle elle doit s'appliquer et la fréquence d'échantillonnage simulé. Cette instruction s'écrit simplement :

```
simulation "Name" timerange "now", "28-02-2018 08:00:00" frequency 1, "MINUTE"
```

simulation est le mot clé pour déclarer une simulation et lui attribuer un nom.

timerange permet de définir la fenêtre temporelle, l'utilisateur devra entrer ici deux dates ou bien le mot clé "now" pour saisir la date actuelle.

frequency défini la fréquence d'échantillonnage, ici 1 suivi de l'unité "MINUTE" signifie que la simulation produira une mesure par minute pour chaque capteurs.

Ces paramètres sont globaux, un utilisateur désirant avoir des paramètres de simulation différent pour des capteurs séparé devra écrire plusieurs simulation. Cela est fait pour garder une cohérence global de la simulation (lors de composition par exemple cela évite d'avoir à interpoler les données de capteurs).

La syntaxe pour la déclaration des capteurs a été faite de manière à être le plus explicite possible et se rapproche du langage naturelle. Un utilisateur ayant connaissance des moyens de modéliser un capteur n'aura ainsi aucun mal à l'utiliser.

```
addSensor "Name" withLaw laws.XXX applyNoise transforms.XXX
```

addSensor permet de rajouter un capteur à la simulation. Ce capteur sera identifié par son nom (Il servira à lui appliquer des transformation, créer des composition mais aussi nommer les données de sortie).

withLaw permet de définir la loi qui régit ce capteur. Une loi est une fonction (au sens mathématique) qui à un instant t associe une mesure (qui peut être de n'importe quel type). Les lois disponibles sont étiquetées laws.XXX. On remplace XXX par la loi désirée.
Parmis ces lois nous disposons de :

- **Identity** qui à une source (CSV, base de donnée) va associer une loi et ainsi permettre de rejouer des données.
- **replayCSV** est un raccourci de la loi précédente permettant d'utiliser l'identity sur une source CSV.
- **polynomial1D** qui va définir un comportement polynomial et dont les mesures seront des nombres réels (ex: la température)
- **polynomialRegression** va permettre d'approximer les données d'une source pour en tirer un comportement polynomial. L'utilisateur choisira le degré du polynôme pour l'approximation. Les mesures du capteur seront des nombres réels.
- **function** permet de définir une fonction du temps que le capteur suivra (la fonction est défini sous forme de lambda expression comme par exemple: $t \longrightarrow t + 1 - 12 * t$) . Ici aussi, les mesures créés sont des nombres réels.
- **random1D** est une loi qui permet de créer un capteur dont les mesures sont issue d'un tirage aléatoire. Elle prend en paramètre deux réels, min et max. Ce tirage suit une loi de probabilité continue sur [min, max]. Min et max étant les valeurs minimale et maximales que les mesures peuvent prendre. Les mesures créés sont des nombres réels.

applyNoise permet d'appliquer des transformations sur les mesures ici de la loi qui régit le capteur. Typiquement, cela permet de rajouter du bruit, de lisser, etc. Un capteur peut contenir plusieurs transformation consécutives. L'ordre d'ajout des transformation est l'ordre dans lesquelles elles sont appliquées (ex: soit 3 transformation t1 t2 t3 et une mesure, $m(t_1, t_2, t_3) \Rightarrow t_3(t_2(t_1(m)))$). Nous disposons des transformations suivantes:

- **random** permet d'ajouter du bruit aléatoire sur toutes les mesures du capteur. L'utilisateur définit son amplitude. Cette transformation ne s'applique qu'aux capteur dont les mesures sont des nombres réels. La valeur du bruit ajouté suit une loi pseudo aléatoire uniforme.
- **partialRandom** permet d'ajouter du bruit aléatoire sur une partie des mesures d'un capteurs. L'utilisateur définit la probabilité pour une mesure d'être affecté par le bruit (a la proportion globale pour un grand nombre de mesures) et l'amplitude du bruit. Ainsi une valeur de 0.5 et une amplitude de 10 ajoutera un bruit aléatoire de ± 5 ($\frac{amplitude}{2}$) sur environ 50% des mesures du capteurs.

Notre DSL permet de dupliquer des capteurs afin de créer des ensemble au comportement similaire. Cela sera utile pour la suite et la composition de capteurs. Pour utiliser cette fonctionnalité il suffit de donner le nom du sensor a dupliquer et le nombre de copies, précédé par le mot clé **generateSet**, de cette manière :

```
generateSet "Name", 12
```

Une fois cela définit, l'utilisateur choisira le format de sortie de la simulation de cette manière :

```
outputTo out.XXX
```

Précédé du mot clé **outputTo**, out.XXX se refere a une sortie parmis celles ci :

- **CSV**: il faudra alors donner le chemin vers lequel enregistrer les fichiers (attention à ne pas donner de nom de fichiers). Si le chemin n'existe pas, il sera créé.
- **Database**: Il prend en paramètre le nom de la base de données ou écrire. La base doit être créé avant l'écriture, elle ne sera pas créé automatiquement.

Les sources sont définie d'une manière identiques aux sorties et sont utiles dans la création de certaines lois de capteur.

```
sources.XXX
```

Nous disposons des deux sources suivantes:

- **CSV** qui prend un chemin de fichier csv (le nom du fichier étant inclu dans le chemin). Les données extraites seront typé dynamiquement. L'algorithme tentera dans l'ordre: d'extraire un nombre entier (Integer), un nombre réel (double), un booléen, en cas d'échec au essais précédant, le type sera String.
- **Database** qui prend le nom de la base et les données à récupérer.

Pour terminer, la syntaxe pour lancer la simulation est la suivante :

```
runSimulation()
```

L'extension

En ce qui concerne notre extension, nous avons choisi d'implémenter celui de la composition de capteurs au sein de notre projet. Nous avons donc la possibilité d'unifier plusieurs capteurs en une seul capteur composite.

Pour ce faire nous l'avons implémenté comme suite. Nous avons créé une interface *CompositionLaw* qui extend *SensorLaw*. Cette interface est implémenté par *SumComposition* et *MeanComposition* qui sont chacun un type de composition qu'on peut appliquer à notre fusion de sensor.

Pour l'utiliser, il suffit d'appeler la fonction **addCompositeSensor** et donner un nom à la composition. Cette composition se fait via des sensors déjà prédéfini. Pour pouvoir définir quels sensors nous allons composé, on utilise le mot clé **fromSensors** qui est suivi de la

liste des sensors à utiliser qu'il faudra définir comme suite : **sensors([**"sensor1**",** "sensor2"**])**. Une fois cela en place, il nous faut choisir le type de composition que nous voulons appliquer en utilisant le mot clé **composeUsing** suivi du modèle à appliquer.

Dans l'exemple ci-dessous, on peut voir comment on créer une composition nommé "composite" qui prend les sensors "light" et "temp" à composer et applique une composition de type "sum" soit la somme des deux sensors qu'on veut composer :

```
addCompositeSensor "composite" fromSensors sensors(["light", "temp"]) composeUsing
compositions.sum()
```

Nous avons donc la possibilité d'unifier nos sensors de cette façon. Un fichier script groovy est présent dans le dossier "syntaxV2" sous le nom de "sumComposition.groovy" pour pouvoir tester notre composition de sensor avec le mode sum.

Exemple concret

M. Chevalier est le nouveau responsable technique de polytech. Il a pour fonction de vérifier que tout le matériel électronique fonctionne mais surtout, il doit monitorer leurs consommation électrique en temps réel. L'école soucieuse des dépenses énergétiques demande à M. Chevalier de s'assurer que les dépenses énergétiques globale journalière ne dépasse pas 1MWh. M.Chevalier, futé, a décidé d'installer des capteurs de consommation sur les prises et appareil de chauffe.

Il veut donc maintenant simuler leurs comportement pour s'assurer qu'ils ne dépassent pas le seuil, auquel cas des actions devront être prise. Il rédige donc le script suivant:

```
simulation "Energy" timerange "28-02-2018 00:00:00", "28-02-2018 23:59:59" frequency 1,
"MINUTE"

addSensor "PriseElectrique" withLaw
laws.polynomialRegression(sources.CSV("mesure.csv"), 3) applyNoise
transforms.random(3.0)

addSensor "Chauffage" withLaw laws.function("t -> -t*t/(100*60*60*24*2) + t/100*4 +
100") applyNoise transforms.random(3.0)

generateSet "PriseElectrique", 200
generateSet "Chauffage", 50

addCompositeSensor "Consommation" withLaw compositions.sum(sensors(["all"]))

outputTo out.CSV("mySimulationDirectory")

runSimulation()
```

Ici, M.Chevalier crée une simulation qui commence le 28 février à minuit et fini la 28 février à 23h59, soit une journée. Il ajoute ensuite deux capteurs. Le premiers est une prise électrique simple qu'il va modéliser depuis un fichier qu'il possède sur son disque dur à l'aide d'une loi polynomiale de degrée 3. Il va appliqué un peu de bruit a cette modélisation pour qu'elle soit plus réaliste.
Il ajoute ensuite un chauffage qui lui est décrit par une fonction périodique du temps (il consomme plus la nuit que le matin car il fait plus froid mais tout les jours sont similaires). Il y ajoute aussi du bruit pour plus de réalisme.
Il crée ensuite des ensemble de capteur pour représenter l'école toute entière.
Pour déterminer la consommation globale, il ajoute ensuite un capteur composite qui donne la somme de tous les capteurs.
Enfin, il décide que ces données seront stocké dans des fichiers csv et il lance la simulation.
Malheureusement, le résultat de cette expérience conclut à un pic de consommation dépassant 1MWh vers 7h du matin. Avec le froid en ce moment, l'école devra prendre des mesures pour diminuer la consommation. Quoique, si les élèves ne sont pas là, les prises consommerons moins.

Analyse critique

Pour commencer, comme nous l'avions expliqué plus haut dans la partie syntaxe de notre document, nous avions utilisé, dans notre DSL, une syntaxe par enchaînement de mots clés renseignés en valeur par des int ou des string. Ce procédé est en soit une facilité pour l'utilisateur final, car le langage devient, de part cette syntaxe simpliste, très facile à prendre en main sous prétexte de bien connaître les mots clés bien évidemment.

Mais nous nous sommes rendu compte que cette façon de faire, nous bloquait dans la phase d'amélioration de notre produit. En effet, nous sommes rendu compte qu'on passait à côté d'élément important qui est l'évolutivité de notre DSL. De plus, on se retrouvait avec un langage Groovy inexploité de notre part, ce qui est dommage car l'un des avantages de faire un DSL embarqué est justement de bénéficier de toute la puissance du langage hôte et profiter de la syntaxe proposée par Groovy.

Nous avons pu, par la même occasion, introduire la gestion des erreurs clients dans notre DSL, ce qui permettra à un utilisateur lambda de ne pas rester bloqué face à une erreur inattendu. La gestion d'erreur syntaxique relevant du compilateur, elle est très compliquée (nécessite de vérifier l'arbre syntaxique) et nous ne l'integrons pas dans notre corrections d'erreur. Nous prenons cependant en compte une partie des erreurs sémantiques que l'utilisateur peut faire (mettre un mauvais *timerange* pour la simulation par exemple).

Conclusion

Le développement d'un DSL nécessite des compétences importantes à la fois en programmation et dans le domaine concerné. En effet, les DSLs sont par définition des langages permettant de répondre à des problèmes spécifique et pour cela, il nous faut bien connaître le domaine et bien cerner le besoin des utilisateurs qui vont le manier. Dans le

cadre de notre projet nous avons pu constater à quel point cerner le domaine (ici les capteurs) était important. Nous avons surtout pu constater qu'il était très complexe de réaliser un DSL intégrant le domaine entier de manière très intuitive pour l'utilisateur. Souvent, des éléments seront implicite dans le langage (ex: les types de données des capteurs, que doit on faire quand la fréquence d'échantillonnage d'un capteur est différente,...). Les utilisateurs auront cependant une courbe d'apprentissage plus rapide, pour les experts du domaine, qu'un langage de programmation. Il ne requiert, généralement, pas de connaissance de langage de programmation pour l'utilisateur final. Cela va permettre de représenter un gain de productivité durant l'exploitation plus important. Cependant, le coût de conception lui dépend du domaine et du type d'implémentation, et peut s'avérer, de ce fait, plus ou moins facile de concevoir un DSL.

Dans notre cas, nous avons utilisé Groovy, ce qui nous a largement facilité la tâche de conception dans un premier temps, en nous affranchissant de la partie <<parsing>> grâce aux *GroovyScripts* et dans un second temps, en profitant d'une syntaxe très proche de Java qui propose notamment une syntaxe plus concise avec le support des listes, des maps par exemple et etc.

Dans la majorité des cas, un DSL tendra à être le plus explicite et le plus spécialisé possible, son gain en productivité y étant lié. Cette particularité (forte spécialisation) a pour effet de bord d'obtenir un langage avec niveau d'abstraction très faible. Le manque d'abstraction du langage le rend difficilement maintenable dans le temps. Autrement dit, un DSL est très coûteux à maintenir. C'est le cas dans notre projet pour lequel nous devions modifier le kernel pour obtenir de nouvelles fonctionnalités.

Pour permettre une modification facile du DSL, il nous a fallu trouver la bonne syntaxe qui convient à faciliter cette tâche et rendre notre produit le plus évolutif possible. On fait ainsi un premier pas vers la maintenabilité.

Enfin, le DSL, dans certaines conditions, profite des avantages cité en limitant ses inconvénients. La condition majeur est de se spécialiser à très petit grain sur un domaine réduit. De cette manière on obtient un DSL légers donc facilement maintenables, portables et réutilisables avec, le plus souvent, une bonne documentation ce qui les rends cohérents et fiables.