Rapport DSL

Sensor Simulation Language

César COLLE
Loris FRIEDEL
Loïck MAHIEUX
Thomas MUNOZ

28 février 2018

Introduction

Ce document présente le langage de simulation de capteur nommé SSL et développé par l'équipe HCS. Son objectif est de simplifier la création de simulation de capteurs à partir de différentes sources de données et d'en permettre une exécution en parallèle. Les résultats de la simulation ainsi généré seront envoyés dans une base de données de type InfluxDB.

SSL est un langage **externe**, basé sur la technologie **ANTLR4** et supporté par **Java** pour la création de son modèle ainsi que son exécution. Nous avons choisi de développer l'extension **Simulation Temps Réel** dont nous parlerons en fin de de document.

Modèle

Voir représentation UML du méta-modèle du langage SSL (figure 1, annexe page 9).

Le modèle comporte cinq grand concepts : les lois, les capteurs, les zones, les applications ainsi que la configuration globale de la simulation.

Une application définie une partie de la simulation exécutable indépendamment et est composée d'instances de zones. Une zone comporte des groupes de capteurs, eux-même définis par une quantité et un type de capteur. Un capteur est régit par une loi et possède divers paramètres de configuration. Une loi définie la manière de produire une valeur à partir d'un timestamp Unix.

Enfin, la configuration globale régit toutes les applications et décide de leur mode de fonctionnement ou encore des paramètres temporels qu'elles utiliseront.

Syntaxe

La syntaxe du langage SSL a été conçue pour être peu permissive, chaque élément doit être correctement indenté et les espacements doivent être parfaitement respectés. La coloration syntaxique ainsi que l'auto-complétion naïve n'est supportée que dans l'IDE Intellij IDEA.

Afin de faciliter la compréhension de la suite du document, nous allons définir le type "**nombre**" qui correspond à un entier ou un flottant, et le type "**tous les types**" qui correspond à une chaîne de caractères, un entier, un flottant ou encore un booléen. Lorsque plusieurs valeurs sont définies au sein d'une même instruction ou d'un même ensemble d'instructions, il est sous-entendu que tous les types de valeurs sont les mêmes.

Définition de lois

Une loi définit la manière de produire une valeur à partir d'un timestamp Unix et se déclare avec le mot clé "law". Toutes définitions de lois possèdent un nom et un type ainsi qu'un ensemble d'instructions de configuration propre à ce type.

Random

La loi aléatoire s'utilise avec le mot clé "random" et permet de produire des valeurs choisies de façon uniformément aléatoire. Il est possible de définir un interval de valeur de type **nombre** ou encore une liste de valeur, qui peuvent être de **tous les types** :

```
law random lawRandInterval {
    values in interval [3.14, 12.9]
}
```

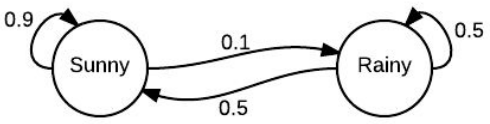
```
law random lawRandList {
    values in list (18, 20, 22, 24)
}
```

Markov

La loi de Markov (utilisé avec le mot clé "markov") permet de décrire une chaîne de Markov, composé d'un nombre fini d'état et leurs probabilités de transition associés. Les états correspondent aux valeurs qui seront produites et peuvent être de **tous les types** :

```
law markov lawMarkov {
    "sunny" -> 0.1 -> "rainy"
    "sunny" -> 0.9 -> "sunny"
    "rainy" -> 0.5 -> "sunny"
    "rainy" -> 0.5 -> "rainy"
}
```

⇒



Dans cette exemple, les états sont des chaînes de caractères et chaque état est séparé par une probabilité comprise entre 0 et 1. La somme des probabilités sortantes de chaque état doit être égale à 1.

Function

La loi de type fonction utilisant le mot clé "function" permet de définir une fonction mathématique qui prend en entrée le timestamp courant en secondes ("x" **uniquement** car c'est la variable que tout le monde utilise dans les fonctions mathématiques, c'est aussi plus lisible et clairement défini dans la documentation du langage) et qui peut produire une valeure de **tous les types.** Les expressions mathématiques sont entouré de '`' et autorise ce que permet la bibliothèque https://github.com/uklimaschewski/EvalEx
Il est possible de définir plusieurs cas, la première expression (la condition) sera alors utilisée pour choisir quelle expression évaluer, à la manière du pattern matching.

```
law function polyMilieuJournee {
    `x%86400 < 32200` => `floor(x/25000000)`
    `x%86400 > 32200` => `abs(-(2*x^2) + 5*x - 1)`
    `x%86400 = 32200` => `0`
}
```

Si l'utilisateur ne souhaite définir qu'une seule expression à évaluer en tout temps, il peut écrire dans la condition soit "TRUE" soit "x" qui donnera la forme standard **x → f(x)**.

```
law function constFromage {
    `x` => "fromage"
}
```

```
law function const42 {
    `x` => 42
}
```

```
law function simpleCarre {
    `x` => `(x/100000000)^2`
}
```

File

La loi de fichier (mot clé “file”) décrit l’utilisation d’un ensemble de valeurs préexistantes en tant que source de données. La loi doit comporter le nom du capteur dont les données seront collectées dans le fichier ainsi que les détails techniques le concernant (c’est-à-dire sa localisation, son type et son adresse). Il est possible ensuite de définir les champs et colonnes qui devront être collectés, dans le cas du CSV on peut définir l’index de la colonne ou bien son nom si il est présent dans l’entête du fichier. Pour les fichiers JSON, seule la définition d’un champ est autorisée.

```
law file lawFileCsv {
    named "sensorName1"
    from local csv "path/to/script/sensorData1.csv"
    using column 1 as time, column 2 as value and column 4 as name
    with linear interpolation restricted to [-1,1]
}

law file lawFileJson {
    named "sensorName2"
    from distant json "http://url.to.script/sensorData2.json"
    using field "x" as time, field "v" as value and field "n" as name
}
```

Finalement, l’utilisateur peut ordonner une interpolation linéaire sur ses données, avec une restriction sur les valeurs ainsi produite (optionnelle).

Définition de capteurs

```
sensor sensorRandInterval {
    governed by lawRandInterval
    period 30m
}

sensor sensorRandList {
    governed by lawRandList
    period 1h
    noise [1, 4]
}

sensor sensorMarkov {
    governed by lawMarkov
    period 1d
}

sensor sensorFunctionMulti {
    governed by lawFunctionMulti
    period 10m
}

sensor sensorFunctionSimple {
    governed by lawFunctionSimple
    period 45s
}

sensor sensorFileCsv {
    governed by lawFileCsv
    period 500ms
}

sensor sensorFileJson {
    governed by lawFileJson
    period 90s
    noise [-0.30, 0.20]
}
```

Une fois des lois définies, il est possible de définir des capteurs qui vont utiliser ces lois. La définition d’un capteur est composée d’une référence à une loi existante, une période (ignorée pour les capteurs basés sur une loi de type **file,** elle peut être omise) qui détermine la fréquence de mise à jour du-dit capteur, puis d’un bruit (optionnel) uniquement utilisable sur les capteurs utilisant des valeurs de type **nombre**. Le bruit doit avoir le même type de valeur que celles produites par la loi utilisée par le capteur.

Les unités de temps disponibles pour la période sont: millisecondes (ms), secondes (s), minutes (m), heures (h), jour (d).

Définition de zones

Pour utiliser les capteurs précédemment déclarés, il suffit de définir des zones (via le mot clé “area”) qui vont contenir des ensemble de capteurs. Chaque groupe de capteur est défini par une quantité et un type de capteur, avec en plus la possibilité de re-définir un bruit pour ce groupe de capteur en particulier.

```
area area1 {
    has 8 sensorRandList
    has 4 sensorFunctionMulti
}

area area2 {
    has 1 sensorMarkov
    has 1 sensorFileJson
    has 2 sensorRandInterval with noise [0.5, 3.5]
}

area area3 {
    has 12 sensorFileCsv
    has 4 sensorMarkov
    has 1 sensorFunctionSimple
}
```

Définition d’application

Finalement, pour produire des applications indépendamment exécutable, il faut déclarer des application en utilisant le mot clé “app”. Sa configuration se résume en l’association d’un type de zone aux noms des différentes instances de ce type qui devront être exécuté.

```
app app1 {
    area1: A1, A2
    area2: B1
}

app app2 {
    area3: C1, C2, C3, C4
}
```

La configuration ci-dessus va donc engendrer la création de deux applications distinctes qui pourront être exécutées en parallèle et dans laquelle, par exemple pour “app1”, on trouvera deux instances de “area1” nommées A1 et A2 et une instance de “area2” nommée B1.

Définition générale

Afin de déterminer les paramètres temporels de l’exécution de la simulation, l’utilisateur peut déclarer un bloc avec le mot clé “global” dans lequel il définit le mode de simulation (**realtime** ou **replay**). Dans le cas du mode replay, il doit déclarer la date de début et celle de fin, tandis que dans le mode realtime il ne peut que définir un offset sous forme de date (qui fera croire que le temps réel est dans le passé ou dans le futur selon la date). Ce bloc est optionnel, la valeur par défaut étant le mode realtime à partir de la date d’exécution du programme.

```
global {
    replay
        start 01/01/2018 00:00
        end 31/01/2018 00:00
}
```

Extension - **E_4: Real Time Simulations**

L'extension choisie, **Simulation Temps Réel**, consiste à exécuter la simulation sans envoyer toutes les données en même temps dans la base de données mais en la remplissant au fur et à mesure en fonction de la fréquence de mise à jour des capteurs à la manière d'un parc de capteurs.

Pour l'implémenter, nous n'avons eu que deux modifications à effectuer :

- ajouter un mot clé dans la grammaire et sa gestion dans le modèle, puis supporter les deux modes d'exécutions dans la bibliothèque du runtime. Pour l'activer, il faut donc préciser dans la configuration globale du script le mot clé "realtime" au lieu du mot clé "replay"
- lors du lancement de l'application, au lieu d'exécuter la simulation de manière séquentielle et sans attente, le programme lance de manière concurrente des tâches récurrentes (interval régulier pour les lois **from scratch**) ou temporisées (interval non régulier pour les lois **file**). Cependant, le métier exécuté reste exactement le même. Cette extension à l'avantage d'être extrêmement peu intrusive dans notre système déjà existant.

Analyse des choix

Syntaxe

A l'origine, nous avions choisi de développer un langage interne reposant sur Groovy. Cependant, nous avions une approche "grammar first" et nous devions faire trop de compromis sur la syntaxe si nous ne voulions pas entamer des modifications qui auraient pu corrompre notre âme. Etant donnée que nous avions déjà écrit la grammaire en pseudo-code nous sommes passé sur ANTLR4, ce qui rendait le remplissage du modèle beaucoup plus simple et les contraintes bien plus faciles à écrire. D'une manière générale, nous voulions un langage extrêmement contraint pour empêcher l'utilisateur d'écrire des scripts non-conformes, ce que permet facilement un langage externe et une grammaire. De cette manière, les scripts sont clairs, uniformes et cohérents, ce qui améliore leur lisibilité et leur maintenabilité, au détriment de la liberté d'expression.

En ce qui concerne la syntaxe, notre objectif principal était de fournir un DSL compréhensible et utilisable par un expert en données mais pas forcément expert en capteur.
C'est pour cela que chaque définition d'élément possède une syntaxe adaptée à sa problématique. Par exemple, nous avons représenté la loi fonction d'une manière naturellement proche de la définition d'une fonction en mathématique, de même pour la chaîne de markov. On perd en cohérence syntaxique entre les différents types de définitions mais on gagne en proximité sémantique avec le métier concerné. D'un manière générale nous voulions cette syntaxe car elle est très proche du domaine et elle reste claire, même si parfois verbeuse (ce qui permet de "lire" le script naturellement).

Toutes les vérifications sont effectuées à la compilation, soit par la grammaire (par exemple sur les contraintes des types de données) ou directement à la construction du modèle, ce qui permet de donner des messages d'erreur clairs à l'utilisateur si son script est invalide.

Modèle

Nous avons choisi de séparer la définition des types d'éléments et leurs utilisations (instanciations). Cela permet de réutiliser les différentes lois, capteurs et zones à la manière d'une classe Java, cela améliore également la clarté du script et sa maintenabilité (principe de séparation des préoccupations)

Toutes les sources de données sont représentées par des lois, même si certaines nécessitent un traitement particulier au remplissage du modèle, comme les lois basées sur des fichiers. De cette manière on distingue clairement les sources de données de leurs utilisations. Concernant l'interpolation linéaire, nous avons choisi de ne supporter que l'interpolation à partir de données d'un fichier, on ne peut donc pas décrire directement un tableau de valeur dans le script. Cela permet de simplifier le contenu du script et de séparer les sources de données de leurs utilisations, cependant cela nécessite d'avoir des fichiers supplémentaire à côté du script.

De plus, nous avons ajouté une fonctionnalité de restriction de valeur dans le cas où l'utilisateur souhaiterait limiter les erreurs d'une interpolation sur ses données.

Nous avons fait le choix de générer une application complètement indépendante en terme de ressources, c'est à dire que lorsque le modèle est rempli, toutes les opérations d'interpolation linéaire ou de chargement des données brutes d'un fichier sont effectuées pour pouvoir générer le code associé et éviter de devoir charger des ressources externes lors de l'exécution de nos applications ainsi générées. Cependant, cela augmente le temps de compilation du script.

Dans notre modèle, il est possible de re-définir un bruit pour un groupe de capteur donné, de cette manière il est possible de simuler le fait que certaines zones soient plus bruitées que d'autre sans pour autant définir des capteurs supplémentaires.

Nous n'avons pas implémenté le support des périodes aléatoires sous forme d'intervalle pour les capteurs, cependant si nous devions l'ajouter cela ne prendrait que quelques minutes (nous évoquerons ce point la dans la partie suivante du rapport).

Pour le support des fichiers JSON, nous aurions aussi pu rajouter la définition du chemin ou se trouve le *JSON array* à lire, nous avons préféré contraindre la forme du fichier pour éviter trop de configuration de la part de l'utilisateur (convention over configuration) cependant, dans ce cas cela aurait été judicieux de mettre une valeur par défaut (c'est à dire le *JSON array* à la racine) et la possibilité de donner un chemin plus précis. Tout comme le support des périodes aléatoires, cela ne prendrait que quelques minutes à implémenter.

Concernant les lois fonctions, si aucune condition n'est vraie lors de leurs évaluations à l'exécution, la valeur par défaut de ce type sera alors renvoyée. Dans le cas ou plusieurs conditions sont vraies, la première à être évaluée sera choisie. C'est à l'utilisateur de s'assurer que sa loi est correctement définie pour son domaine.

Pour simplifier la génération du code, nous avons implémenté une *model-aware generation*. Nous avons donc une bibliothèque de runtime développée, testée et utilisable indépendamment.

Exécution

Le parsing d'un script .ssl est peu performant car certaines règles de grammaire qui nous permettent de vérifier toute la cohérence des types dans une déclaration de lois sont peu efficaces.Dans le cas de SSL, il s'agit d'effectuer une simulation après la compilation, c'est donc sans importance de devoir attendre quelques secondes/minutes pour compiler (dans le cas de script énorme).

En mode temps réel, le programme utilise des threads pour simuler la parallélisation des capteurs, ce qui est en Java assez rapidement limitant, l'utilisateur devra donc dimensionner son support matériel en fonction de sa simulation en partant du pire scénario qui est un thread par capteur.

Nous fournissons un script permettant de construire les images Docker des applications directement exécutables uniquement à partir d'un script .ssl donné.

Nous avons choisi d'utiliser Docker pour exécuter nos applications, en plus de permettre d'exécuter manuellement le jar compilé. Cela permet d'éviter à l'utilisateur d'avoir à installer l'environnement Java sur les machines qui vont exécuter la simulation, et si il souhaite les déployer sur plusieurs machines, il pourra aisément utiliser un cluster Swarm. Il n'y a qu'un docker-compose.yml à créer afin de configurer comment on souhaite déployer nos applications (fixer une applications sur une machine précise par exemple) et de déployer la stack sur le cluster Swarm.

Prise de recul

Domaine

Lorsqu'on développe un DSL, l'objectif est d'obtenir un produit adapté aux experts du domaine pour maximiser leur productivité tout en restant dans les limites du contexte du-dit domaine.
Cependant, dans le cas de SSL le sujet était large, ouvert et les contraintes minime, de plus nous n'avions pas à proprement parler d'expert du domaine ni de client réel avec qui discuter du DSL. A cause de cela, nous avons ainsi pu facilement confirmer que notre produit suivait les spécifications du sujet mais de façon beaucoup moins évidente qu'il restait pertinent dans le domaines de la simulation des capteurs (par exemple, est-il vraiment intéressant de pouvoir définir des zones réutilisable ? Ou encore est-il nécessaire de permettre à une loi aléatoire de renvoyer des chaînes de caractères et pas seulement que des nombres ?)

Il était tout autant délicat de limiter les fonctionnalités que nous ajoutions puisque notre DSL est externe et nous pouvions faire ce que nous voulions très simplement. Finalement, construire un DSL sans un 'D' clairement borné (de notre point de vue), c'est un exercice très intéressant.

Extensibilité

Dans le cas de SSL, nous offrons certains type de lois utilisable pour définir le comportement de capteurs. Cependant, leur faible nombre limite grandement l'utilisation du DSL et nous pensons qu'ils serait intéressant d'offrir la possibilité de définir sa propre loi. En revanche, cet ajout aurait un coût extrêmement élevé mais une valeur ajouté tout autant importante, dans le cas ou on ne souhaiterais pas avoir un développeur à disposition pour implémenter la moindre loi qu'un expert souhaiterait utiliser. Par contre, ce n'est pas intéressant dans tous les DSL et il faudrait certainement une durée de vie et un ensemble d'utilisateurs du DSL très élevé pour justifier un tel coût.

Conclusion

Finalement, faire un DSL c'est comme les tartes aux abricots a la pate feuilleté coeur de blé Herta, c'est bon, mais c'est chaud !

Annexe

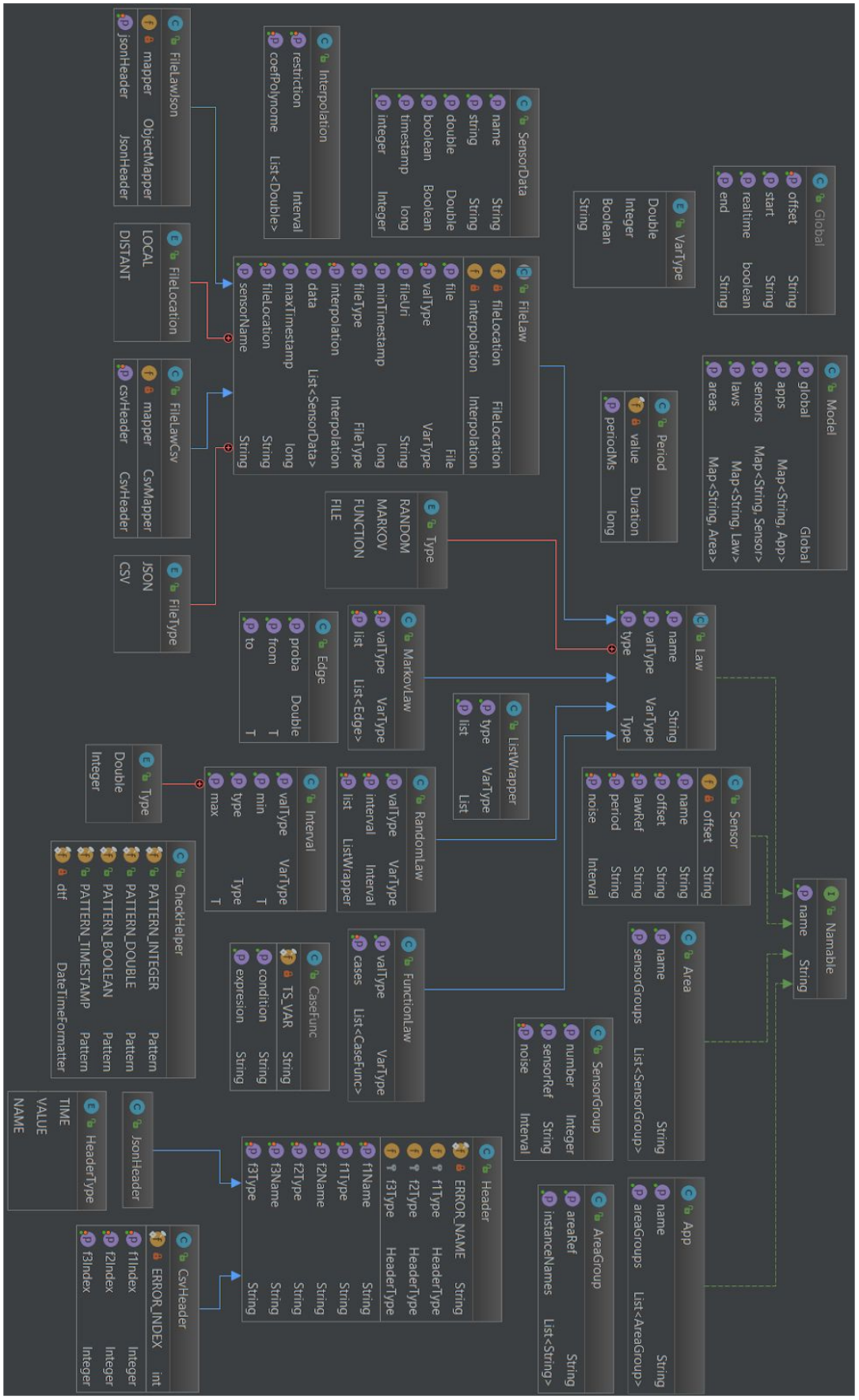


Figure 1 : représentation UML du méta-modèle du langage SSL