# SenSiLang - Sensor Simulation Language

*DSL*

*Carlier Maxime - Chevalier Mathias - Jungbluth Günther*

POLYTECH NICE SOPHIA

## Introduction

Le monde des capteurs et des objets connectés (auxquels nous ferons référence en parlant de l'*Internet of Things - IoT*) est en expansion exponentielle depuis quelques années, et se révèlent être un secteur clé dans l'évolution technologique des Smart Cities ou des Smart Grid, qui elles mêmes sont des entités clés dans leurs domaines respectifs: respectivement l'optimisation des flux des citadins et des services aux citoyens ainsi que l'optimisation énergétique. La gestion des données est cependant une problématique réelle, puisque ces données doivent être à minima mises à disposition, et plus tard doivent être interprétées pour devenir des connaissances et des informations. La donnée brute n'a en effet qu'une petite valeur, et c'est à travers la masse et la connexion d'informations que la valeur ajoutée s'amplifie grandement. Un besoin qui découle directement de ces problématiques serait donc de pouvoir simuler, jouer des scénarios choisis, rejouer des ensembles de données pour finalement essayer d'anticiper les problèmes imbriqués et cachés en profondeur.

C'est donc à cette fin que nous avons implémenté et design un langage spécifique à ce domaine: *SenSiLang* qui est donc un *Sensor Simulation Language*. Bien évidemment, nous ne couvrons pas l'ensemble des domaines inhérents aux problématiques présentées dans le paragraphe précédent, mais nous nous attaquons à la partie simulation. Ce document présente les caractéristiques de notre implémentation.

# Domaine & Choix Architecturaux

*Voir diagramme en annexe.*

Nous avons dans un premier temps choisis d'utiliser une classe abstraite de haut niveau "AbstractSensor" afin de modéliser un comportement communs à toutes les classes modélisant un capteur quelle que soit leurs sources de données. Cette classe abstraite définit un attribut ID permettant d'identifier le capteur, et un champ Mutator qui permet de définir au niveau le plus haut la façon dont les mutateur agissent sur les valeurs renvoyés par un capteur. Cette classe abstraite possède deux méthodes permettant pour l'une de définir le contrat à remplir pour être qualifiable de capteur, a savoir, pour un timestamp arbitraire, de renvoyer un Number (Int, Double, Float, etc.) ou une absence de valeur (null). la seconde méthode, quant à elle sert à définir au niveau hiérarchique le plus haut, la logique permettant de retourner une valeur selon la présence ou non d'un mutateur (ex: bruit) en appliquant la méthode de l'interface Mutator qui fait varier selon un comportement définis par une implémentation la valeur obtenus par le contrat de AbstractSensor.

FunctionnalSensor définit le comportement commun à des capteurs dis fonctionnel, CAD des capteurs capables pour n'importe quelle valeurs en entrée de calculer soit même la valeur (qu'elle existe ou non). FunctionnalSensor définit ainsi un unique attribut Function qui permet à ses classe fille de venir écrire un objet de ce type afin de remplir le contrat. Ainsi, FunctionSensor, permet à l'utilisateur de définir une lambda arbitraire pour la production de valeur tant que cette lambda capture un argument (même s'il est abandonné ensuite). RandomSensor de son côté, définit un capteur basé sur une lois aléatoire dont l'utilisateur peut définir les bornes. Un typage de haut niveau Number nécessite lors de la création par l'utilisateur d'un FunctionalSensor (RandomSensor par exemple) de définir non seulement les bornes, mais en plus de spécifier un *subtype* de Number afin de raffiner le comportement attendu. Ainsi définir un capteur aléatoire avec des borne 1;5 et un type Integer, entraînera la production des valeurs 1;2;3;4 alors que la même création avec un type Double, retournera des valeurs telles que $1 \leq x < 5$ et $x \in \mathbb{R}$.

Un des choix intentionnels que nous avons fait, est pour les capteur d'interpolation, de déléguer l'ensemble de la logique d'interpolation à une librairie externe (ci-nommée Apache Commons-Math). La raison étant que ce type d'opération peut s'avérer difficile à implémenter sans effleurer les problématique liée à la convergence de tels algorithmes et qu'il est ainsi préférable de ne pas s'en préoccuper. La conséquence sur notre Architecture est la présence d'une séparation hierarchique entre le InterpSensor, et les autres FunctionalSensor (Random et Function). En effet, bien que le comportement d'un capteur d'interpolation soit identique à celui d'un capteur Fonction après que l'interpolation est été effectuée ; Il n'hérite pas de la même classe du au fait qu'il n'est pas possible de "ramener" l'objet final de l'interpolation (ie : UnivariateFunction), à celui que nous utilisons nous (Function).

Dans le cas d'un capteur Markovien, nous avons fait le choix de proposer une approche plus intuitive pour initialiser les éléments. En effet, nous ne prenons pas directement la matrice stochastique mais nous offrons la possibilité de définir les états indépendamment puis d'ajouter ensuite des transitions d'un état vers un autre avec une probabilité souhaitée. Bien évidemment, une fois le capteur instancié, la matrice est vérifiée afin d'éviter d'avoir un état avec une somme de probabilités d'états sortants supérieure à 1.

De plus, notre algorithme permet de compléter les lignes dont la somme serait inférieure à 1 et où la probabilité de boucler sur l'état ne serait pas définie par l'utilisateur, par:

*1 - SUM(Probabilités états sortant autres que la boucle sur l'état propre)*

L'objectif de cette implémentation est donc de simplifier l'utilisation pour des utilisateurs qui ne seraient pas directement familiers avec la science derrière les capteurs Markoviens (i.e capteurs modélisés par un modèle de Markov), et donc de permettre à plus d'utilisateurs d'utiliser SenSiLang. L'ensemble de l'API supportant les fonctionnalités de Markov se veut pouvoir être générique, l'implémentation n'a cependant pas été complétée mais l'effort pour faire cette modification serait faible compte tenu de la structure du code. Il serait donc tout à fait envisageable d'ajouter le support de la matrice stochastique directement plutôt que l'ajout d'états puis de transitions, notons tout de même que la couche DSL serait tout de même d'une légère complexité. Il serait aussi possible de mettre en place une composition en offrant le support pour les états de ne plus représenter une valeur statique mais une fonction comme le ferait un FunctionSensor.

Le type de capteur suivant, ReplaySensor dont l'objectif est de fournir un support pour rejouer des données, utilise un découpageen deux interface pour séparer les responsabilité de : récupérer les donnée (FileAdapter) et interpreter les données (Coder). L'avantage d'un point de vu architectural ici, est que le découplage est total et que la flexibilité d'utilisation est maximale. L'utilisateur (expert on le rappel) peut ainsi s'il le souhaite sous seul contrainte de remplir les contrats définis par les deux interfaces, créer son propre FileAdapter et son propre Coder permettant d'interpréter le format du fichier.

# Syntaxe

## I. Définir un Capteur Aléatoire

```
randomSensor "my-sensor", 0, 10
```

**randomSensor** : Définit le type de capteur à créer.
**"my-sensor"** : Définit l'ID du capteur ainsi que la référence vers cet objet (cf : Mutator)
**0, 10** : Définit les borne inférieure et supérieure pour la portée de la loi aléatoire.

Nous mettons ici à profit le typage statique des nombres 0 et 10 (Integer) qui permet à l'interpréteur Groovy d'inférer la surcharge qui sera utilisée pour l'instanciation de ce capteur. On rappelle que comme expliqué dans la partie Domaine, les capteur aléatoire travaille sur une plage, et sur un type (Integer, Double) qui change les donnée qui sont renvoyée. Ici le compilateur infère la surcharge utilisant le type Integer, mais si les valeurs auraient été 0.0d et 10.0d cela aurait été la surcharge Double qui aurait été utilisée.

## II. Définir une plage simulation

```
simulate 0,300,50
```

**simulate** : Mot clé pour l'exécution de la simulation

**0 :** Le timestamp de départ de la simulation

**300** : Le timestamp limite que la simulation ne pourra pas excéder

**50** : Optionnel, le pas de la simulation (1 par défaut)

Cette première syntaxe permet d'exporter le modèle sémantique et de l'exécuter dans une simulation borné. Les résultats s'affiche dans la console sous forme d'un tableau croisé. Ex :

| Time | replaySensor | replaySensor_noise |
|---|---|---|
| 0 | -728.0 | -679.1273723989298 |
| 50 | N/A | N/A |
| 100 | -727.0 | -783.5738393589968 |
| 150 | N/A | N/A |
| 200 | -730.0 | -721.5814796354321 |
| 250 | N/A | N/A |
| 300 | -728.0 | -665.0952034930283 |

## III. Définir un capteur fonctionnel

```
Function test = { x ->
    if (x % 2 == 0)
        0
    else
        1
}
functionSensor "my_function" with test
```

**Function test = { x -> ...}** : Définit la fonction chargée de modéliser le capteur. x joue le rôle du timestamp pour lequel il faut retourner une valeure.

**functionSensor :** Définit le type de capteur à créer

**"my_function" :** Définit l'ID du capteur et sa référence

**with :** mot clé permettant de passer en paramètre la fonction a utiliser pour le capteur courant

**test :** Référence vers un fonction définie précédemment

## IV. Utiliser un capteur Markovien

```
markovSensor "my-markov"
markovState "my-markov" add 0, "zero", 13.84
markovInit "my-markov"
markovTransition "my-markov" add 0, 1, 0.23
markovStart "my-markov"
```

Nous ne présenterons ici pas les caractéristiques qui parlent d'elles-mêmes et qui reprennent des points citées dans d'autres capteurs précédemment.

Les commandes `markovState`, `markovInit`, `markovTransition` ainsi que `markovStart` sont les commandes permettant les appels définis dans l'API métier. Leurs fonctionnalités respectives sont donc l'ajout d'état, la déclaration du fait que tous les états souhaités aient été instanciées, puis l'ajout de transition et le fait que toutes les transitions souhaitées aient été déclarées. L'objectif est de fournir un cadre assez précis afin de ne pas perdre l'utilisateur et de ne pas lui permettre une liberté totale qui amènerait à de nombreuses erreurs.

Le mot-clé `add` précède la définition respectivement des états et des transitions. Les états sont définis comme suit:

```
0, "zero", 13.84 -> id, description, valeur
```

Et les transitions:

```
0, 1, 0.23 -> id initial, id final, probabilité
```

## V. Définir un capteur permettant de rejouer une série

```
replaySensor "replaySensor",Integer.class forFile
"./sensilang-api/src/main/resources/Bike1.csv" asCrossedCSV 10
```

**replaySensor** : Définit qu'il faut construire un capteur replay

**"replaySensor"** : L'ID du capteur et sa référence

**Integer.class :** Le type de donnée à rejouer. Peut être utiliser pour faire des troncatures si on spécifie un type Entier alors que la série est Décimale.

**forFile :** mot clé permettant de définir que le fichier est localisé sur le système de fichier

**"./sensilang-api/..."** : Spécifie le chemin vers le fichier.

**asCrossedCSV** : Définit que le fichier est encodé sous forme de tableau croisés.

**10** : Le numéro de colonne de la série à rejouer

## VI. Définir un mutateur permettant de faire varier le résultat d'un capteur

```
replaySensor "replaySensor",Integer.class forFile
 "./sensilang-api/src/main/resources/Bike1.csv" asCrossedCSV 10
replaySensor "replaySensor_noise",Integer forFile
 "./sensilang-api/src/main/resources/Bike1.csv" asCrossedCSV 10

addNoise 0.1,"replaySensor_noise"

simulate 0,300,50
```

**addNoise :** Mot clé permettant de définir qu'on souhaite ajouter un mutateur de bruit

**0.1** : l'amplitude du bruit. cela signifie que pour toute valeur rejouée x, sa valeur finale après application du bruit sera comprise entre (1-0.1)*x ≤ x ≤ (1+0.1)*x

**"replaySensor_noise" :** référence vers un capteur définis précédemment qui permet d'identifier à quel capteur sera appliqué la mutation

### VII. Cloner un capteur

```
randomSensor "my-sensor", 0, 10 clone 5
```

**clone :** Mot clé permettant de cloner un capteur. Dans l'exemple ci-dessus on copie 5 fois le capteur.

### VIII. Insérer les résultats sour Grafana

Nous avons utilisé le driver java de InfluxDB pour ajouter les différentes exécutions. Ces exécutions sont sauvegardés lors de l'appel de la méthode save. En voici un exemple:

```
simulation = export 0,300,50
save simulation
```

Les séries induites des simulations finissent, dans le temps, à l'instant présent. La configuration de Grafana se fait manuellement à travers l'interface graphique.

Il est alors possible de définir des dashboards en fonctions des simulations en sélectionnant les capteurs de ces simulations.

# Mise en abîme

## Motivations

Le contexte et le domaine dans lequel s'inscrit l'*IoT* et ses problématiques fournit un cadre où insérer un langage spécifique a du sens et une motivation réelle. Nous nous en rendons en fait compte de manière très intuitive. Tout ce contexte est en réalité très complexe, car il allie des niveaux de granularité nombreux allant du grain très fin qui représente un capteur unique aux plus gros grains qui seraient des réseaux de réseaux... de réseaux de capteurs. Ces nombreuses couches nécessitent donc une abstraction afin de les rendre accessibles à des personnes qui seraient des experts d'un domaine d'application (par exemple la gestion de l'eau), mais qui ne disposent pas forcément des compétences informatiques nécessaires à l'implémentation d'un projet. Notons ici que nous considérons que ces experts possèdent des connaissances liés aux capteurs, motivant ainsi nos différents choix d'implémentations (langage Groovy, états pour la chaîne de Markov, ..). De plus, même les personnes familières avec le développement pourraient utiliser notre langage afin d'aller plus vite et d'éviter la mise en place potentiellement fastidieuse d'une solution similaire. *SenSiLang* s'inscrit donc dans ce cadre en essaye de rendre accessible, relativement simplement, la simulation de réseaux de capteurs. La partie visualisation est aussi intéressante afin de pouvoir avoir un retour lisible et intuitif, le tout sur une interface web.

### Le champ des possibles

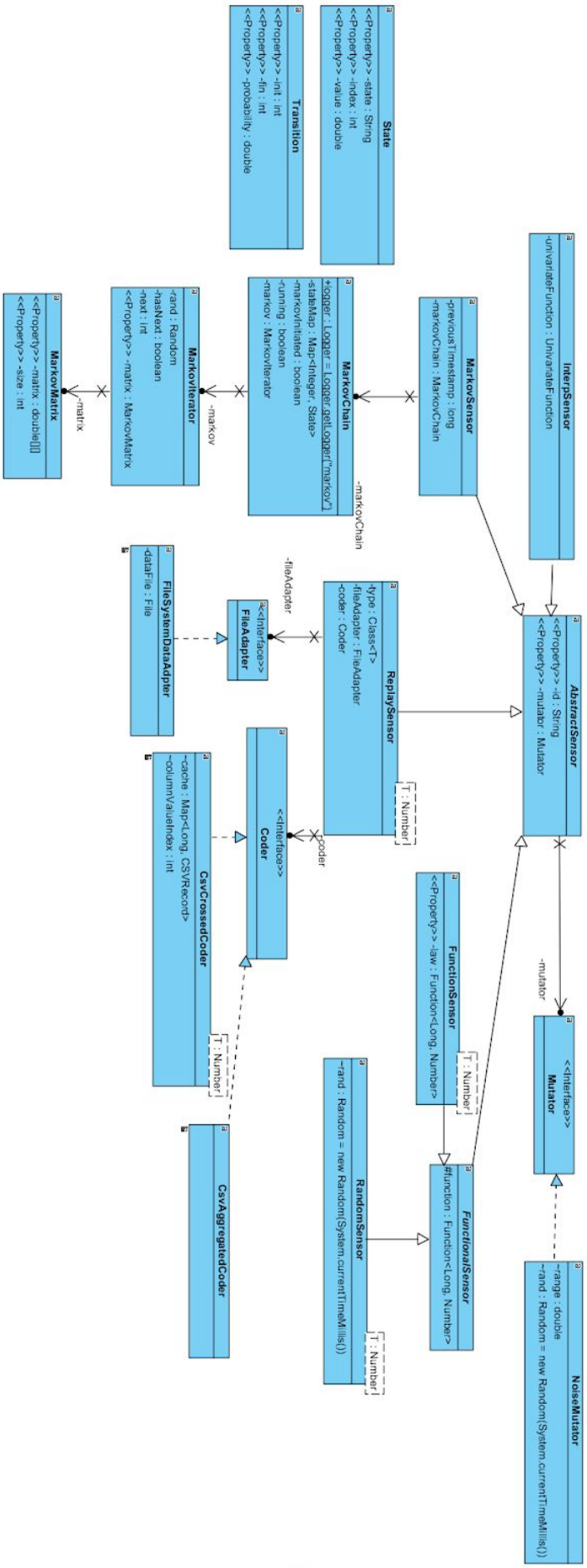
Les possibilités d'évolutions du DSL sont titanesques. En effet, *SenSiLang* permet de rejouer des simulations, mais il permet aussi de lier des capteurs entre eux afin d'avoir une simulation à plus grande échelle. De nombreuses fonctionnalités pourraient être ajoutées qui devraient pouvoir être implémentés sans changer la logique déjà mise en place. La syntaxe devra bien évidemment évoluer, et la couche applicative ainsi que le DSL lui-même devront changer quelque peu, mais cela devrait pouvoir se faire sans aucune régression fonctionnelle.

### Limites

Néanmoins, la complexité liée à la systémique et donc au fait que ces réseaux résultent de la composition de sous-réseaux est comme nous avons pu le voir, immense. De plus, les acteurs sur le marché se livrent probablement à une guerre concurrentielle qui pourrait s'opposer aux desseins de ces technologies, et ils pourraient donc aisément se montrer réticents à la généralisation de leurs propres modèles. De plus, c'est un champ d'application qui est au final extrêmement large, et comme nous le savons, l'objectif principal d'un DSL est évidemment de rester spécifique. Malgré tout, à chaque version du langage proposé, nous pouvons déjà imaginer à la fois les demande et les "Et si on ajoutait ça? C'est pas grand chose et ça apporterait plein de choses!". Concrètement, se limiter et fournir un effort de maintenance et de réponses aux besoins risque d'être très complexe et limiter le potentiel du langage.

## Conclusion

Finalement, la création d'un langage tel que *SenSiLang* est une tâche complexe, nécessitant des capacités de réflexion et des capacités techniques non négligeables. Il est par exemple important de noter que le langage n'est pas aussi simple que nous l'aurions souhaité, mais nous n'avons pas réussi dans un temps raisonnable à implémenter certains éléments syntaxiques complexes. Cependant, dans l'ensemble, la preuve de concept est réalisée et permet la visualisation de résultats de simulation sur une interface web dédiée, la personnalisation de ces simulations et des capteurs ainsi que l'exécution de ladite simulation. *SensiLang* gagnerait donc bien à évoluer, son domaine métier est très intéressant à approcher et est évidemment porteur dans le contexte d'aujourd'hui. Le dernier point à approcher est aussi la possibilité d'optimisation de performances que nous n'avons pas mis au centre du *scope* et qui pourrait donc être améliorée dans le futur.


State
<<Property>> -state : String
<<Property>> -index : int
<<Property>> -value : double
Transition
<<Property>> -init : int
<<Property>> -fin : int
<<Property>> -probability : double
InterpSensor
-univariateFunction : UnivariateFunction
MarkovSensor
-previousTimestamp : long
-markovChain : MarkovChain
MarkovChain
+logger : Logger = Logger.getLogger("markov")
-stateMap : Map<Integer, State>
-markovInitiated : boolean
-running : boolean
-markov : MarkovIterator
-markovChain
MarkovIterator
-rand : Random
-hasNext : boolean
-next : int
<<Property>> -matrix : MarkovMatrix
-matrix
MarkovMatrix
<<Property>> -matrix : double[][]
<<Property>> -size : int
-fileAdapter
<<Interface>>
FileAdapter
FileSystemDataAdpter
-dataFile : File
ReplaySensor
-type : Class<T>
-fileAdapter : FileAdapter
-coder : Coder
T : Number
coder
AbstractSensor
<<Property>> -id : String
<<Property>> -mutator : Mutator
-mutator
<<Interface>>
Coder
CsvCrossedCoder
-cache : Map<Long, CSVRecord>
-columnValueIndex : int
T : Number
CsvAggregatedCoder
FunctionSensor
<<Property>> -law : Function<Long, Number>
T : Number
RandomSensor
-rand : Random = new Random(System.currentTimeMillis())
T : Number
#function : Function<Long, Number>
FunctionalSensor
<<Interface>>
Mutator
NoiseMutator
-range : double
-rand : Random = new Random(System.currentTimeMillis())