DSL
Rapport final
Sensor Simulation Lab

BELHASSEN Issam
BALDE Thierno
SUIGNARD Thomas

Plan:

I.Introduction

La société dans laquelle nous vivons évolue, l'informatique envahit de plus en plus notre paysage. Le lien entre notre monde et le monde informatique est fait grâce à des capteurs. On en trouve partout, dans les voitures, dans les parkings, etc... Une fois les données recueillies par ces capteurs, ces informations sont utilisées par les infrastructures de prochaines générations telles que Smart Cities ou Smart Grid. Développer un middleware dédié aux réseaux de capteurs est une tâche complexe qui nécessite d'être testé. Or, il est impossible et surtout insensé de tester ces middlewares en conditions réelles en étant connecté aux milliers de capteurs. C'est là que nous, et surtout notre dsl, intervenons. Notre but est de développer un outil pour générer ces données afin d'effectuer ces tests. Plusieurs questions sont apparues avant la conception de ce DSL, quel type de DSL voulons nous, quel langage voulons nous utiliser, quel langage voulons nous implémenter ?
Nous avons choisi d'implémenter un DSL externe en MPS, le langage de code généré que nous avons choisi c'est java.

II.Description de notre langage

1. Diagramme de classe

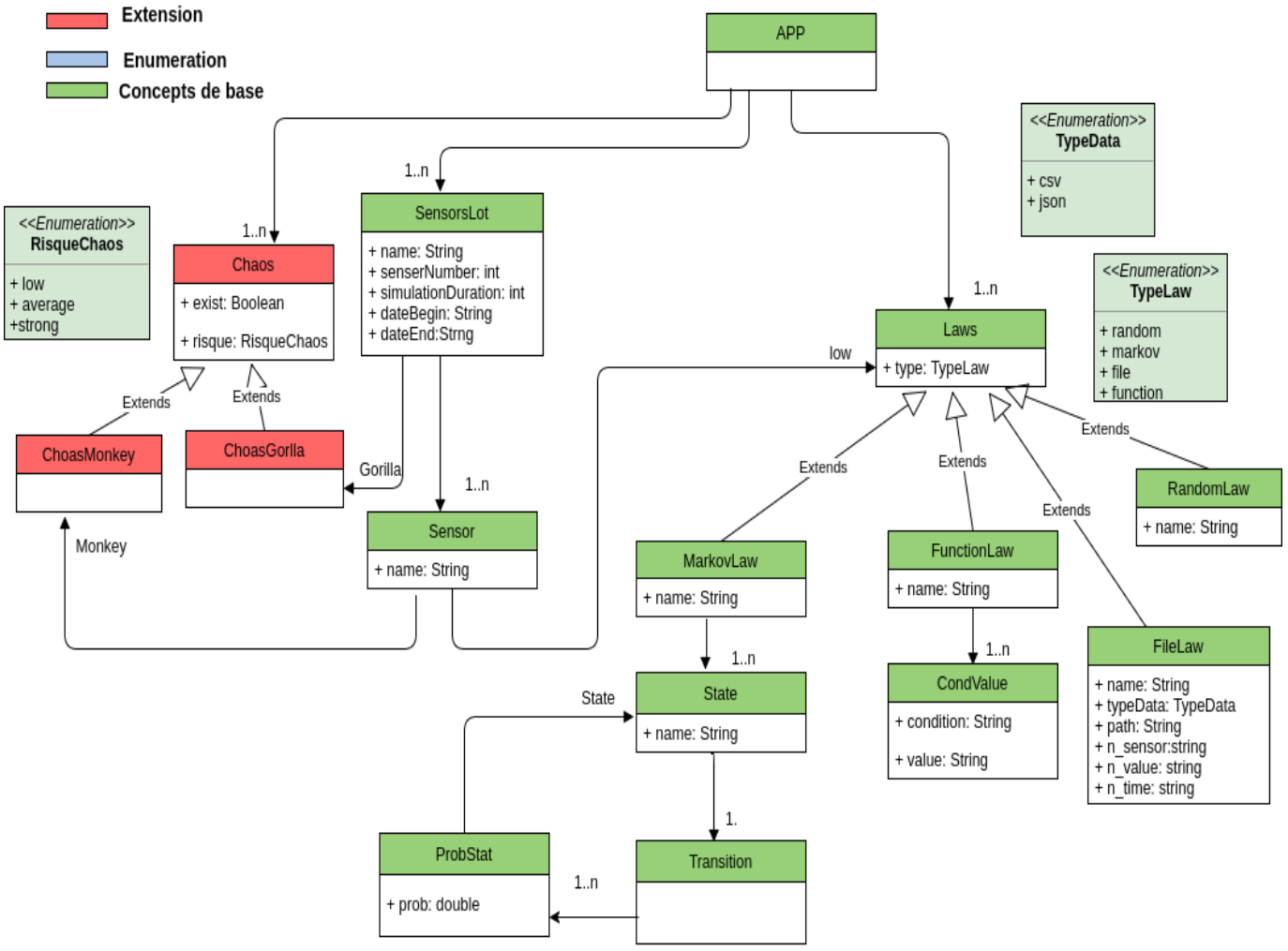


figure1 : diagramme de classe

Voici un diagramme de classe qui représente l'arbre syntaxique de notre langage. On peut voir trois classes principales, SensorsLot, Chaos et Laws. Nous avons choisi cette implémentation parce qu'elle nous donne la possibilité de simuler plusieurs sensors lot en ayant plusieurs laws.

Quatre classes héritent de la classe Law, MarkovLaw, FunctionLaw, FileLaw et RandomLaw. Ce sont nos 4 lois possibles. L'attribut commun est TypeLaw qui est un type énumération, qui représente

une loi de notre langage. Pour la loi FileLaw nous avons un type énumeration (csv, json), nous avons choisi de regrouper les sources de données csv et json dans une seule loi, Il aurait été un peu lourd de faire de nouvelles lois qui auraient hérité de FileLaw alors qu'il y en a que deux qui se ressemblent énormément, seul l'extension du fichier utilisé change. Pour implémenter l'extension nous avons rajouté deux classes qui héritent de la classe Chaos, ChoasMonkey et ChoasGorilla avec l'énumération RisqueChaos qui représente le type de risque.

2. Syntaxe concrète

Notre langage se décline en trois parties. La première partie est utilisée pour la déclaration des lots de sensors, la deuxième est pour utiliser les lois que les sensors utilisent et la troisième partie concerne l'extension chaos monkey gorilla.
Notre syntaxe concrète est peu verbeuse et facile à appréhender. Grâce à la projection nous avons un choix de représentation libre.

Voici un exemple :

```
app App {

  sensor lots :
    sensorLot parkingRandom sensorNumber 6 Low  randomlaw Duration 12 from 16/02/2018 00:00:00 to 19/02/2018 03:00:00
    sensorLot jardinCsv sensorNumber 12 Low  csvLaw Duration 10 from 16/02/2018 00:00:00  to 19/02/2018 01:00:00
    sensorLot bureauJson sensorNumber 6 Low  jsonLaw Duration 10 from 16/02/2018 00:00:00  to 19/02/2018 04:50:00
    sensorLot salleFunction sensorNumber 14 Low  functionLaw Duration 10 from 16/02/2018 00:00:00  to 21/02/2018 00:00:00
    sensorLot garageMarkov sensorNumber 20 Low  markovLaw Duration 10 from 16/02/2018 00:00:00  to 19/02/2018 00:00:00
    sensorLot <no name> sensorNumber <no sensorNumber> Low  <no low> Duration <no simulationDuration> from <no dateBegin> to <no dateEn
  lows :
    randomLow : randomlaw
    fileLaw csvLaw from /home/user/Bureau/data/dataCsv.csv format csv with s: 1 v: 8 t: 0 with offset 0
    fileLaw jsonLaw from /home/user/Bureau/data/dataJson.json format json with s: sensorName v: value t: time with offset 4
    functionLow functionLaw Expresion { x<1 : 2              }
                                      x>= 1 && x<=3 : x^2-3
                                      x>3 : abs(-2*x)
    markov markovLaw {
      state sunny : [ sunny 0.9 ] [ sunny 0.05 ] [ sunny 0.05 ]
      state runny : [ runny 0.4 ] [ runny 0.4 ] [ runny 0.2 ]
      state cloudy : [ cloudy 0.4 ] [ cloudy 0.5 ] [ cloudy 0.1 ]
    }
  chaos :
    Gorilla ? true risque average
    Monkey ? true risque strong
}
```

figure2 : syntaxe concrète

- La loi randomLow modélise une simulation avec des valeurs choisies de manière aléatoire. Elle est décrite de cette façon:

```
<default> editor for concept RandomLow
  node cell layout:
    [> randomLow : { name } <]
```

Pour cette loi nous avons besoin seulement de définir son nom, nous ne voyons pas d'autres paramètres à définir.

➢ La loi fileLaw modélise une simulation avec des valeurs issus d'un fichier. Nous proposer deux types de fichiers, csv et json. Elle est décrite de telle façon :

```
<default> editor for concept FileLaw
  node cell layout:
    [> fileLaw { name } from { path } format { typeData } with s: { n_sensor } v: { n_value } t: { n_time } with offset { offset } <]
```

Nous avons choisi les variables t, s et v comme des string, ce choix nous permet de regrouper les deux types de data csv et json dans une seule ligne, pour csv on passe le numéro de colonne(exp. 1,3,8), alors que pour json on passe le nom de champs (exp. sensor,value,time) à chercher dans le fichier.

➢ *La loi functionLaw modélise une simulation de fonctions, chaque valeur est calculé par l'application d'une fonction tout en évaluant une condition.*

```
<default> editor for concept FunctionLow
  node cell layout:
    [> functionLow { name } Expresion { (/ % poly %                /) } <]
                                      /empty cell: <default>
```

```
<default> editor for concept CondValue
  node cell layout:
    [> { condition } : { valeur } <]
```

Nous avons choisi de faire hériter CondValue hérite de FunctionLaw, de cette façon nous pouvons ajouter autant qu'on voudra comme lignes de fonctions.

functionLaw <name> Expression { <list of conditions valeurs> }

➢ La loi markovLaw modélise une simulation issue d'une chaîne de markov.

Grâce au lien de parenté entre le concept de markov et states nous pouvons définir plusieurs states pour la loi markov.

```
<default> editor for concept Markov
  node cell layout:
    [/
      [- markov { name } { -]
      [- ---> (/ % states %              /) -]
                /empty cell: <default>
      [- } -]
    /]
```

Ainsi, cette projection nous permet de passer une transition pour chaque état.

```
<default> editor for concept State
  node cell layout:
    [> state { name } : % transitions % <]
```

De même, pour la transition de l'état nous définissons les couples target-probabilité qui représentent les probabilités des états.

```
<default> editor for concept Transition
  node cell layout:
    [> (> % coupleProb % /empty cell: <default> <) <]
```

```
<default> editor for concept CoupleProb
  node cell layout:
    [> [ ( % target % -> { name } ) { prob } ] <]
```

➢ Pour les lot de sensors, nous définissons le nom, le nombre de sensors, puis la loi associés aux sensors, la durée, dateBegin et dateEnd. La durée totale de simulation est calculée à partir de la multiplication de “simulationDuration” par le temps entre chaque itération.(exp. 10 * 5 seconde)

```
<default> editor for concept SensorLot
  node cell layout:
    [> sensorLot { name } sensorNumber { sensorNumber } Low (> % sensors % /empty cell: <default> <) Duration { simulationDuration } fi
```

Pour associer une loi aux sensors nous avons cette représentation

```
<default> editor for concept Sensor
  node cell layout:
    [> ( % low % -> { name } ) <]
```

3. Extension: Noise, choas Monkeys & chaos Gorille

Nous avons choisi d’implémenter les extensions suivantes à notre DSL : les chaos.
Le monkey qui intervient auprès des différents Sensors de façon aléatoire avec différents type de risque. Le chaos gorille agit sur les lots de sensors avec la même logique que le monkey.
L'utilisateur à la possibilité de définir l'existence ou pas de chaos avec un niveau de risque qui représente la probabilité d’action du chaos monkey sur les sensors et du chaos Gorilla pour les lots de sensors.

- Gorilla

Gorilla simule une panne réseau globale et stop la simulation de sensors lots immédiatement.

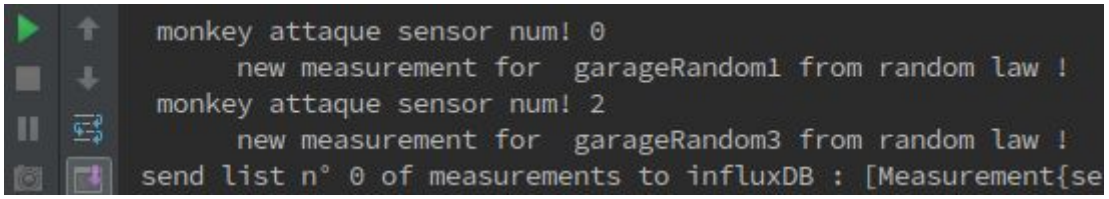
  - risque low: Pour une durée de simulation d, il y a 1/(2d) chance que gorilla agit
  - risque average: Pour une durée d, il y a 1/d chance que gorilla agit
  - risque strong: Pour une durée de simulation d, il ya 1/(d/2) chance que gorilla agit.

- Monkey

Monkey simule une panne temporaire ou définitive d’un sensor et fait perdre le résultat de sensor défectueux.

  - risque low: Pour un nombre de sensor n, il y 1/(n/4) chance de nombre de sensor défectueux.
  - risque average: Pour un nombre de sensor n, il y 1/(n/2) chance de nombre de sensor défectueux.
  - risque strong: Pour un nombre de sensor n, il y 1/n chance de nombre de sensor défectueux.

exp. Attaque Monkey avec un risque strong: sur 4 sensors, on a 2 qui sont défectueux. La liste de mesures envoyées à influxDB contient des valeurs nuls pour les sensors 0 et 2.



exp. Attaque Gorilla avec un risque strong, la simulation est interrompue sans finir 10 lists parmis 12.

```
        new measurement for  parkingFunctон2 from function law ! :
         iff(x<1,2,x>3,abs(-2*x),x>= 1 && x<=3,x^2-3)=1.0 when x = 2.0
        new measurement for  parkingFunctон3 from function law ! :
         iff(x<1,2,x>3,abs(-2*x),x>= 1 && x<=3,x^2-3)=1.0 when x = 2.0
send list n° 2 of measurements to influxDB : [Measurement{sensorName='
Gorilla attack !!! we can't finished 10 lists from our 12 lists

Process finished with exit code 0
```

le problème avec l'annulation de tâche est qu'on peut tomber sur l'état “Waiting” d’un thread. Donc quand Gorilla agit, l'arrêt du thread ne sera pas immédiat car il faudra attendre qu'il repasse dans l'état Running afin de réévaluer la condition de la boucle while et pouvoir l'arrêter.
Pour résoudre le problème, nous avons utilisé le mécanisme d'interruption. Chaque thread possède un statut d'interruption qui indique si une demande d'interruption sur ce thread a été faite ou non. Gorilla demande donc l'interruption d'un thread via la méthode interrupt.
L'avantage d'utiliser ce principe est que, lorsqu'un thread est en attente et Gorilla impose son interruption, il va automatiquement sortir de cet état et se stopper.

III. Code généré et résultats

Règles de réduction

Pour la génération de code java, MPS suit un principe de map/reduce, chaque concept est associé à des règles décrivant quel fragment de code associé sera généré et utilisant quelles propriétés des noeuds accessibles.
On applique les règles de réduction sur ces concepts et nous assurons un code généré de lots de sensors avec les bons sensors et leur loi associées.
Nous avons choisi de générer du code Java. Le code généré doit ensuite être utilisé dans IDE qui supporte Java (un simple projet maven).

- **Random law**

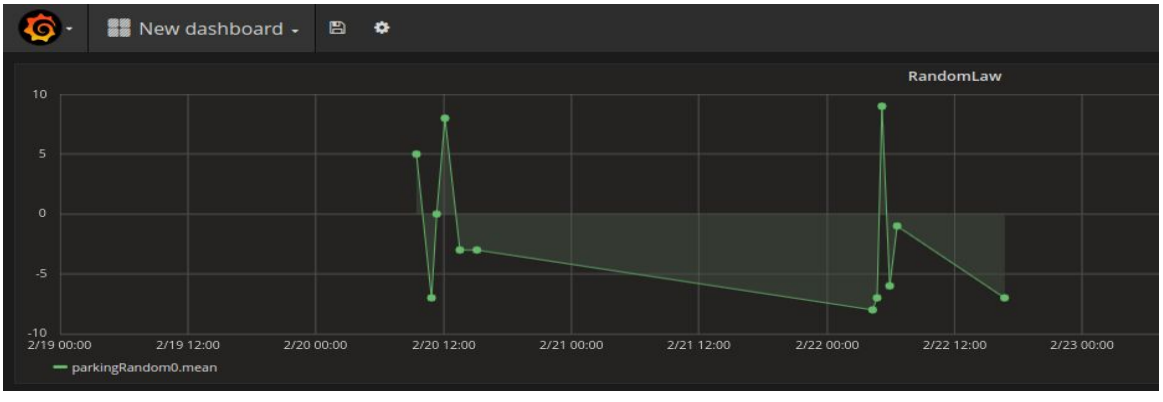


Figure 1 : graphe issu d’une simulation random

Les valeurs et le time sont aléatoires.
*Les noms de sensors sont incrémentales avec le préfixe de nom de lot de sensors (ex. nous simulons 2 sensors dans un lot de sensors appelé **garage**: les noms de sensors sont appelés respectivement **garage0**, **garage1**.*

- **Identification des sources de données csv et Json**

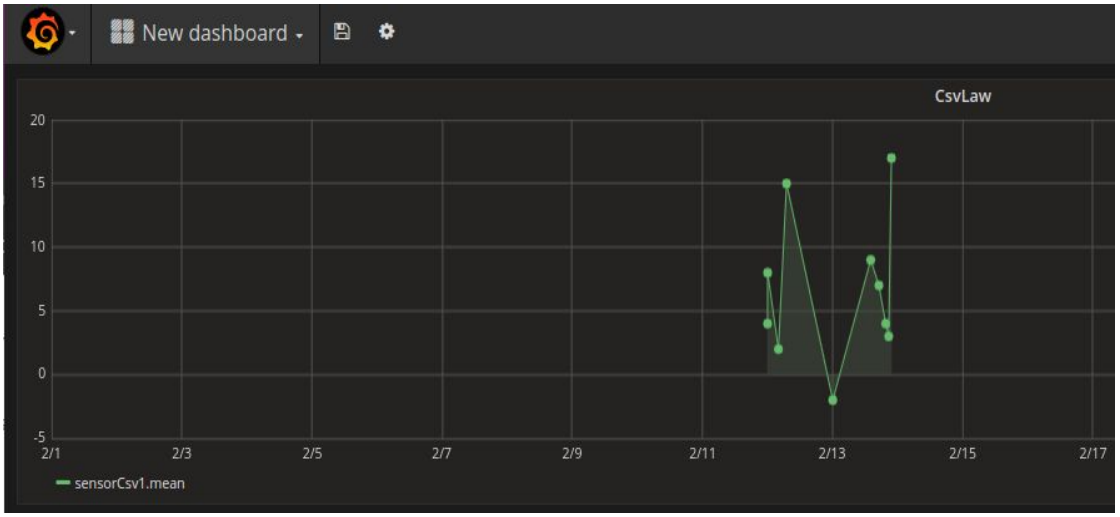


Figure 2 : graphe issu d'une simulation liée à un fichier csv

Les valeurs sont extraites à partir de fichier csv, l'utilisateur indique dans la définition de la loi l'emplacement de triplet sensor, value et time.

Figure 3 : graphe issu d'une simulation lié à un fichier json

Les valeurs sont extraites à partir de fichier json, l'utilisateur dans la définition de loi indique le nom de champs de triplet sensor, value et time.

- **Function law**

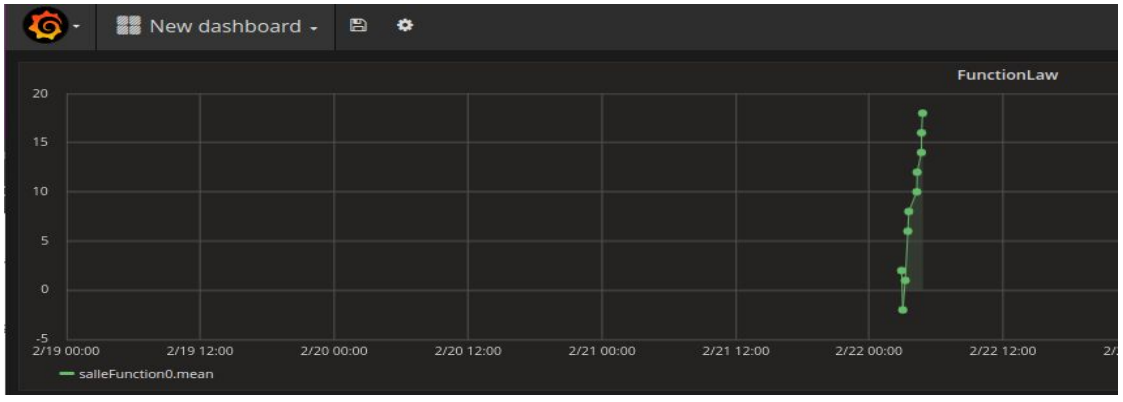


Figure 4 : graphe issu d'une simulation Function

Nous avons utilisé une librairie java pour faciliter nos calculs de fonctions. Nous pensons que cette technique rend nos calculs plus génériques et efficaces, nous avons choisit la librairie mXparser.

- **Markov low**

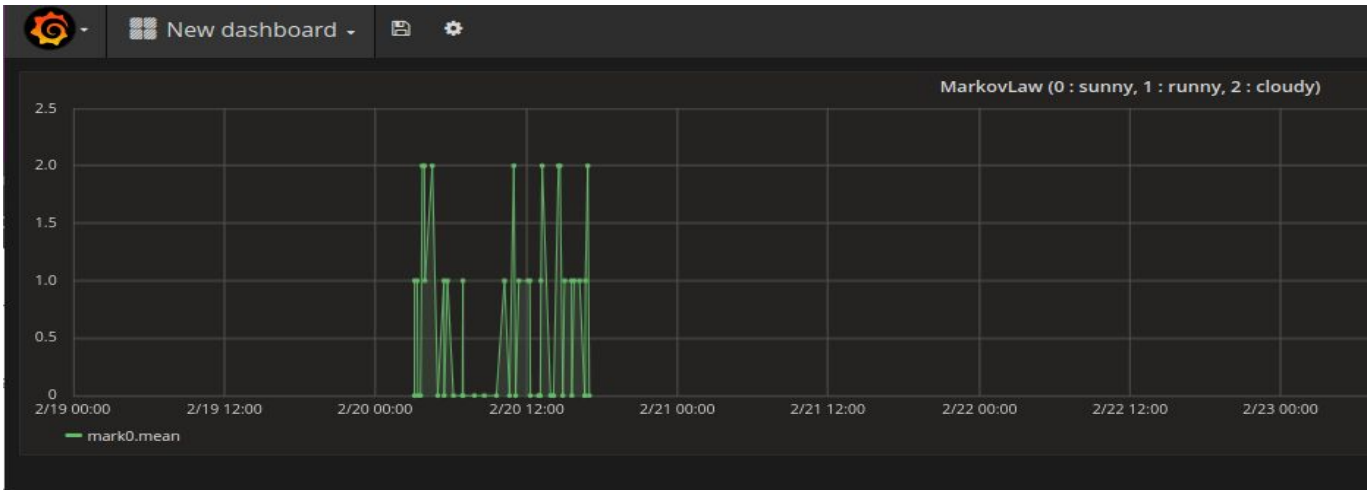


Figure 5 : graphe issu d'une simulation Markov

Nous avons utilisé la librairie mockNeat afin de faciliter le calcul de probabilité de chaque état.

IV.Analyse critiques

Nous sommes évolutifs dans notre domaine, dans le sens qu'on connait notre domaine qui est la simulation de lots de sensors. Nous pensons que notre architecture est extensible puisqu'elle couvre plusieures possibilités de simulations avec une facilité de faire les map-reduces entre les différentes couches de concepts, ce qui nous permet de générer le code java correspondant aux simulations voulues.

Notre système est déjà adapté pour supporter des nouvelles lois de simulations.

Nous avons choisi de générer du code java afin de profiter de sa portabilité (dans la mesure où la compilation et l'exécution de simulation demandent seulement un simple projet maven) et la richesse de ses librairies (pour le calcul de fonctions mathématiques et des probabilités), de plus java nous permet de monter en charge par la création des threads qui nous assurent une exécution simultanée de tâches. Notre DSL assure la simulation de chaque lot de sensors dans un thread isolé, de cette façon nous garantissons une simulation rapide et performante, en plus java nous permet d'assurer nos requêtes de stockage dans influxDB à travers un driver bien défini.

Le choix d'un DSL externe MPS est dû à notre connaissance de la problématique de projet, nous savons que la difficulté de ce projet n'est pas la sémantique de lois et sensors et valeurs retournés mais plutôt la manière de relier les différents concepts entre eux et la manière de chercher l'information exacte en naviguant et appliquant les règles de map-reduces. Nous pensons que MPS nous a aidé à répondre à cette problématique. Grâce à notre expérience avec MPS, nous avons pu rapidement mettre en place un DSL fonctionnel, cette rapidité de développement est dûe à la courbe d'apprentissage descendante de MPS. Maintenant qu'on connait MPS, nous pouvons surmonter tous les problèmes techniques que nous rencontrons. MPS nous permet aussi de mieux présenter nos différentes lois (ex. les expressions de la loi function) et nous permet d'avoir un choix de représentation libre. Un autre point fort de MPS est qu'il nous permet d'avoir un développement conditionné par l'éditeur (ctrl+space & ctrl enter).

Par conséquent, nous pouvons empêcher l'utilisateur de commettre des erreurs syntaxiques (ex. proposer dans le sensor Lot une liste de lois déjà définies, choisir le format csv ou json, ...). Cependant, nous ne gérons pas les erreurs sémantiques au moment de la génération du code (ex: l'url du fichier source est erroné ou n'existe pas), il nous semble que notre choix de génération de code ne nous permet pas d'assurer ça, nous aurions pu faire ça si nous avions choisi d'exécuter la simulation au moment de run de notre DSL.

Parmis les points négatifs de notre DSL c'est la perte de vérification de code métier qu'on génère. Puisqu'on passe par un langage intermédiaire pour créer du code java, ce problème est dû par notre choix de langage externe. Pour limiter ce risque, la solution aurait été de suivre le principe de la méthode de développement Test-Driven Development (TDD) qui écrit les tests avant le code. Donc nous avons suivie ce principe en développant le code java avant de l'avoir généré par MPS, une fois ce code vérifié et ses résultats testés, il suffit de s'occuper de la manière de sa génération depuis MPS (depuis la syntaxe concrète jusqu'à la génération de code). En faisant ses tests d'une manière itérative, nous aurions été sur d'avoir générer du code correct (sauf exceptions dues à des erreurs sémantiques).

Pour aller un peu plus loin dans notre discussion, cette vérification est faite dans la phase de développement, d'une autre manière ce problème de code vérifié sera transparent pour l'utilisateur final, dans la mesure où on sera dans notre domaine, et la génération de code est dirigé par des éléments du scope bien définis tels que la définition des lois et les lots de sensors.

V.DSLs in general in the context of SSL

La simulation de capteurs est un problème qui s'adresse à des informaticiens mais aussi à des experts comme des météorologues, des climatologues, des physiciens, etc...il est donc important pour de telles branches de métiers - mais également pour des informaticiens - d'améliorer leur productivité et la communication avec les différents experts.

L'utilisation d'un DSL prend tout son sens car il augmente l'accessibilité à l'utilisation d'une simulation de capteurs. En effet, l'intérêt de tels simulations est de pouvoir effectuer des comparaisons entre les différentes simulations. Il est très compliqué pour un non initié d'aller modifier les variables en durs dans un langage qui n'est pas adapté. Une fois le DSL créé, il n'y a plus qu'à se concentrer sur les données produites et non sur la production de données. Donc cette approche permet d'améliorer la productivité puisque les DSL sont beaucoup plus expressifs dans leur domaine et offrent une simplicité et lisibilité du problème. Ce problème est exprimé à un niveau d'abstraction qui permet aux experts de comprendre, valider et modifier les détails de leur domaines.

Un DSL est un outil très pratique mais le choix de créer ou non un DSL est souvent déterminé par son utilité futur et par sa complexité. En effet, il n'y a aucun intérêt à créer un outil peu utile et complexe à implémenter. Les informaticiens ont récolté tellement de données à travers le temps, l'heure est peut-être plus à l'analyse qu'à la collecte de données car nous sommes dans une société où tout est connecté et tout doit être prévisionnel : de la météo à la consommation d'énergie en passant par le nombre de places sur un parking. C'est pourquoi un outil tel que le nôtre prend toute son importance et gagnerait à être plus développé.