SSL : rapport

Équipe C : Robin Alonzo, Antoine Aubé, Mathieu Mérino

Ce document présente le travail effectué par l'équipe C dans le cadre du projet de DSL "SSL". Il présente le modèle sur lequel se base ce DSL, sa syntaxe concrète, notre extension et son implémentation, puis effectue une critique de nos choix dans le cadre de ce projet. Il s'achève par une discussion rétrospective sur les DSL dans le contexte de la simulation de réseaux de capteurs.

Modèle/Syntaxe abstraite

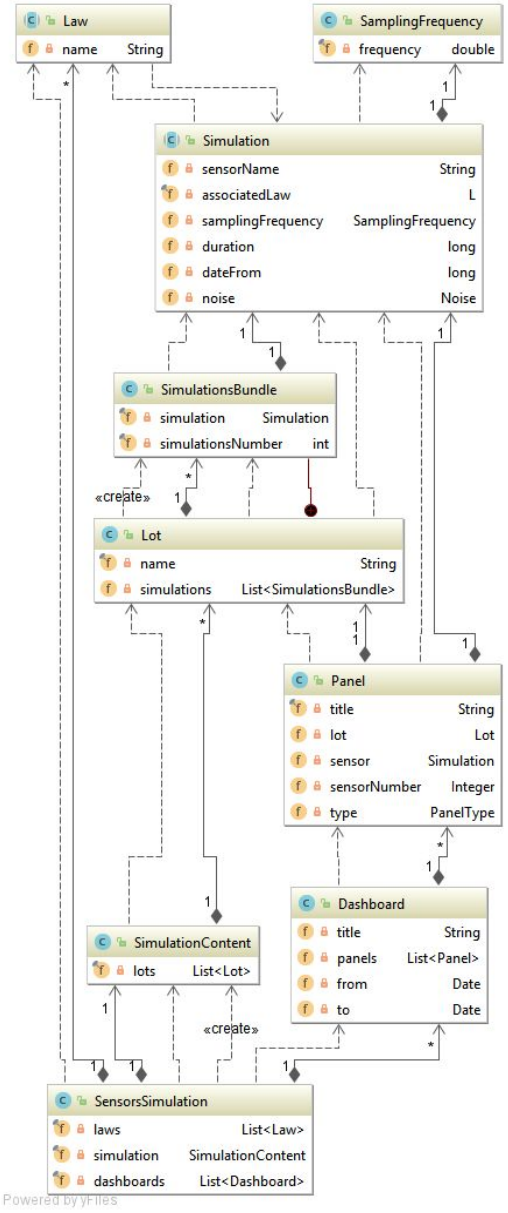

Figure 1 - Syntaxe abstraite de SSL

Syntaxe concrète

Notre langage permet de définir des simulations de réseaux de capteurs.

L'utilisateur définit dans un premier temps les lois que ses capteurs vont suivre. Les lois disponibles et paramétrables sont : une loi aléatoire (choix au hasard d'un nombre sur un intervalle, d'un élément d'un tableau), une loi qui suit une interpolation polynomiale d'un ensemble de points, une loi pour laquelle nous définissons des fonctions mathématiques sur des intervalles de temps.

Nous considérons que les replays sont des lois particulières, ils sont donc décrits avec une syntaxe similaire ; l'utilisateur peut rejouer des sets de données sérialisés en JSON ou en CSV, en spécifiant les colonnes ou champs adéquats et le nom du capteur à rejouer.

Il définit ensuite ses simulations. Les capteurs sont regroupés en lots, que nous avons pensé pour représenter des localisations géographiques (quand bien même ce pourrait aussi être des regroupements sémantiques, ...). Un capteur a un nom et suit une loi avec un ensemble de paramètres (notamment il faut indiquer la durée de la simulation, le nombre de capteur de ce type, la fréquence à laquelle sont générées des mesures).

L'extension que nous avons choisi permet de configurer un tableau de bord. Après avoir défini ses simulations, l'utilisateur peut créer un tableau de bord qu'il peuple comme il le souhaite : il ajoute des graphiques et des tableaux, chacun est associé à un capteur simulé.

Ci-suit un exemple qui montre ce que notre DSL peut faire, et avec quelle expressivité.

```
laws {
  law 'temperature csv' is replay('samples/day.csv') {
    fetch 'temperature' whose values are Integer

    times in 1
    values in 2
    sensors in 3
  }

  law 'parking place occupancy' is random('Available', 'Taken')

  law 'occupancy' is function(0..24) {
    returning Integer

    when 0..4 then {t + 2}
    when 4..8 then {10}
    otherwise {t * 2}
  }

  law 'interpolated temperature' is interpolate(0..24) {
    x 0   y 1
    x 7   y 3
    x 12  y 20
    x 19  y 17
  }
}

simulation {
  lot ('School') {
    contains 3 sensors 'temperature csv' following 'temperature csv' parameterized {
      during 20.minutes
      sampleEvery 1.second
      from '25/04/2017 15:00'
    }
```

```
    contains 5 sensors 'parking place' following 'parking place occupancy' during 2.hours
sampleEvery 1.minute from '25/04/2017 15:00'

    contains 1 sensor 'occupancy' following 'occupancy' parameterized {
      from '25/04/2017 15:00'
      period 1.hour
      sampleEvery 10.minutes
      during 1.day
      noise (-2..3)
    }
  }
}

visualization {
  dashboard ('Complete Simulation') {
    from '25/04/2017 15:00'
    to '25/04/2017 16:00'

    table ('Place') {
      lot 'School'
      sensor 'parking place'
      number 2
    }

    graph ('Occupancy') {
      lot 'School'
      sensor 'occupancy'
      number 0
    }
  }
}
```

Extension : *Visualization dashboards* (E_2)

L'extension que nous avons choisi permet à l'utilisateur d'avoir configuré des tableaux de bord avec les mesures générées.

Pour l'utilisateur, cela se définit après avoir décrit les lois et les simulations. Il indique quelle simulation doit être affichée et sous quelle forme. Ces figures sont affichées sur Grafana, notre outil de visualisation.

En terme de code, cet ajout n'a pas engendré de grandes modifications. Le modèle, la syntaxe et la validation sont triviaux ; l'exécution utilise des paramètres que nous passons au lancement du programme.

Par soucis d'abstraction et pour ne pas mélanger les préoccupations techniques et le métier, nous n'avons pas de référence à Grafana dans le DSL (localisation du serveur, clé d'API).

Nous n'avons pas poussé très loin les personnalisations des visuels via le DSL par manque de temps (le code existant prouve que nous pouvons déjà personnaliser ces visuels, cela dit) et pour ne pas avoir un langage trop spécifique aux possibilités offertes par Grafana (et donc ne pas nous restreindre si nous souhaitons changer d'outil de visualisation).

Analyse critique de nos choix

Un DSL embarqué, développé en Groovy

Nous avons réalisé ce DSL en Groovy et Java ; il s'agit donc d'un DSL embarqué.

Le choix de réaliser un langage embarqué s'est imposé pour les raisons suivantes :

- Le temps disponible pour développer le DSL nous a paru être un facteur limitant et un argument pour développer un langage embarqué (aussi bien pour développer le MVP que pour arriver au bout du projet). Rétrospectivement et avec notre connaissance actuelle du projet, il nous semble que la limitation de temps ne nous aurait pas empêché de développer un DSL externe équivalent (nous l'avouons, c'était une mauvaise raison).
- Suite au DSL ArduinoML, nous avons expérimenté les deux approches et nous nous sentions plus confortables avec un DSL embarqué. Nos connaissance et expérience de cette approche nous ont paru plus solide que pour l'approche externe, c'est pourquoi nous avons préféré renforcer nos compétences dans cette voie plutôt que de nous imposer la difficulté d'outils que nous maîtrisons peu.
- Certaines fonctionnalités du langage donnent une grande liberté à l'utilisateur (par exemple, la loi "fonction mathématique" pour laquelle il faut pouvoir exprimer toutes les opérations arithmétiques). Développer ces fonctionnalités dans un DSL externe serait coûteux alors qu'elles sont "gratuites" dans un DSL embarqué. Ce choix est renforcé par le fait que les utilisateurs de ce DSL ont une expérience de la programmation (e.g. Alice dans le sujet) et qu'ils vont pouvoir tirer parti de bibliothèques pour exprimer des réseaux de capteurs plus complexes.
- Nous ne nous sommes pas imposé une syntaxe concrète pour le langage ; nous avons construit les modèles avant de nous poser la question de comment l'exprimer, et comme nous avions trouvé moyen de tout exprimer de façon satisfaisante en Groovy, la liberté offerte pour un DSL externe ne nous était pas utile.

Nous avons choisi Groovy comme langage hôte ~~car le cours nous en a fait une bonne publicité et~~ car nous avions une expérience satisfaisante de ce langage en tant que langage hôte de par les possibilités qu'il offre pour la syntaxe concrète (*closures*, expression des durées, etc.) et le fait qu'il fonctionne sur la JVM (ce qui inclut l'interopérabilité avec Java que nous avons utilisé pour le modèle la validation et l'exécution, les bibliothèques disponibles pour nous simplifier le développement, etc.). En particulier pour SSL, nous avons apprécié que Groovy offre un moyen simple d'exprimer les durées, et qu'on aie pu utiliser une bibliothèque pour communiquer avec InfluxDB (plutôt que de construire les requêtes à la main).

Un DSL déclaratif en cascades

Notre DSL est déclaratif : l'utilisateur décrit ce qu'il veut dans sa simulation. Cela nous semble avoir peu de sens d'avoir l'approche impérative avec ce langage dans lequel l'utilisateur veut décrire son réseau de capteurs.

Nous avons “compartimenté” les parties du langage pour lui apporter une structure visuelle et simplifier la relecture du code par la suite. Ces compartiments sont implémentés avec des *closures* en Groovy et le “morceau de langage” disponible au sein de ces closures est accessible grâce à la mécanique de délégation. Cette approche nous a permis de modulariser notre définition du langage (pas tout mettre dans la classe qui étend `Script`) et d’avoir un meilleur contrôle sur son contenu (méthodes accessibles différentes selon le type de loi, etc.).

Notre approche en deux temps (décrire les lois utilisées puis décrire les simulations en utilisant les lois et des modifications appliquées dessus) limite la réutilisabilité du code. Nous pourrions souhaiter décrire des lots de capteurs et les composer pour constituer des zones, par exemple (en considérant par exemple qu’un lot est un ensemble de capteurs branchés sur un objet et que cet objet existe en plusieurs exemplaires à divers endroits et en diverses quantités). Pousser autant la réutilisabilité ne s’est pas imposé lors de la création du langage car nous n’avions pas imaginé de situation dans laquelle ce serait pratique ; cela étant, notre conception nous permettrait d’évoluer facilement vers cette réutilisabilité.

Du code bien responsabilisé

L’organisation du code que nous avions expérimenté pendant ArduinoML (séparer en modules le modèle, la syntaxe concrète, la validation et l’exécution) nous avait apporté satisfaction : cela facilite la lecture du projet car ces modules sont indépendants, sauf du modèle bien entendu, et car cela sépare les responsabilités (nous ne voulions pas de notions d’exécution ou de validation dans le modèle, par exemple).

Nous avons reproduit une organisation semblable pour SSL et avons observé les même bienfaits (quand bien même cela nécessite un effort supplémentaire du fait de la variabilité logicielle pour les lois : rassembler tout dans le modèle aurait permis de profiter du polymorphisme, mais cela aurait rendu le code peu maintenable ; notre approche utilise le schéma de conception *Visitor*, qui nous permet de tenir compte de la variabilité logicielle sans tout mélanger).

La prise en charge des erreurs

Nous avons distingué trois types d’erreurs qui peuvent survenir :

- L’utilisateur écrit du code Groovy erroné.
- L’utilisateur décrit un modèle incohérent ou incomplet.
- Un problème survient durant l’exécution du modèle.

Dans le premier cas, nous considérons qu’il s’agit d’un problème de compilation et nous ne le prenons pas en charge pour l’instant (trop coûteux pour peu de valeur). Une évolution permettrait à l’utilisateur de savoir quelle est l’erreur et où elle se situe dans le code (mais nous pensons que ce serait difficile à faire avec nos choix de technologies).

Pour les erreurs d’exécution, nous levons des exceptions et un message d’erreur s’affiche sur la console si elle survient.

Nous avons approfondi la prise en charge des erreurs liées au modèle. La validation se fait avec un visiteur (dans le sens du schéma de conception *Visitor*) qui traverse le modèle décrit par le script et qui constitue un rapport contenant les erreurs et les *warnings* contenus dans le modèle. De cette façon, l'utilisateur a connaissance immédiatement de l'ensemble des corrections à apporter.

L'implémentation de la mécanique de visiteur est un peu coûteuse à mettre en place (créer les interfaces, définir quel élément est visitable, implémenter la visite pour chaque modèle) et pose un problème de volume de la classe du visiteur (qui grandit en même temps que le nombre de modèle), mais offre des avantages de maintenance (à l'ajout d'un modèle, il est impossible de compiler tant que tout ce que cela implique - évolution de l'interface du visiteur, implémentation dans chaque visiteur - n'ait été ajouté, donc nous avons un suivi "fiable") en plus d'apporter une grande plus-value à l'utilisateur (car il est agaçant de corriger les erreurs une par une et de ré-exécuter pour trouver la prochaine erreur).

Un DSL prêt à évoluer

Certains choix que nous avons fait sont motivés par des évolutions futures que nous pourrions apporter au DSL :

- Les lots de capteurs servent uniquement pour le nommage, pour l'instant. Nous les avons ajoutés en prévision d'implémenter l'extension #1 avec le Chaos Gorilla qui peut couper un lot entier de capteurs pendant un temps.
- Nous avons pensé l'exécution comme des tâches à exécuter, lesquelles génèrent des mesures. Ces exécutions pourraient être faites sur plusieurs processus et sur la durée, nous permettant d'implémenter une génération en temps réel (extension #4).

Il y a deux points que nous n'avons pas exploré par manque de temps et car ils ne rentrent pas vraiment dans le sujet, mais que nous aurions souhaité implémenter si le temps nous le permettait.

D'abord, nous souhaitons améliorer la réutilisabilité des morceaux de code. Nous pensons à ajouter une notion de localisation géographique qui contient des lots. Nous ne l'avons pas fait dans cette version car il nous paraît peu pertinent que plusieurs zones contiennent des lots de capteurs à l'identique. Nous pensons à ajouter la notion de variables dans la composition des lots qui permet de moduler le nombre d'un certain type de capteurs, l'amplitude du bruit, etc. afin de pouvoir réutiliser la description d'un même lot dans plusieurs localisations (il suffira d'adapter la valeur des variables).

D'autre part, nous réfléchissons à généraliser les lois, élaborer un métamodèle des lois d'une certaine manière, de façon à pouvoir exprimer les lois que nous avons défini jusqu'ici avec ce métamodèle mais également permettre aux utilisateurs du DSL (ou d'autres acteurs) d'enrichir le DSL sans avoir à modifier le code source du langage. Nous pourrions alors ajouter des types de lois comme dans un système de plugins.

Discussion sur les DSL pour SSL

Au terme de ce projet, nous avons pris du recul par rapport à l'élaboration d'un DSL dans le cadre du projet SSL.

L'ingénierie orientée par les modèles s'est révélée être une manière simple d'approcher la simulation de réseaux de capteurs : il est facile de modéliser les concepts qui constituent une simulation, faire le lien entre les concepts et leur exécution, … L'approche des DSL en découle et ajoute une abstraction au code qui permet à un expert du domaine qui ne connaît pas l'architecture du projet d'écrire des scripts qui simplifient son travail.

Cela étant, notre approche est biaisée car nous ne sommes ni n'avons pu consulter un expert du domaine qui pourrait être utilisateur de notre DSL. Les spécifications nous montrent ce que contient le modèle, mais il nous manque à savoir comment le destinataire du langage va utiliser le langage, ce qui normalement devrait diriger notre construction de la syntaxe concrète. Nous pensons qu'il est important pour un DSL qu'il s'adapte au métier, pas que le métier s'adapte au DSL ; nous pensons par exemple à une approche pour formuler le réseau de capteurs, des "évidences" pour l'expert qui du coup deviennent des redondances dans le langage, etc. Un exemple de DSL auquel nous avons eu affaire et que nous pensons qu'il aurait pu être une simplification radicale de son domaine s'il avait eu une syntaxe adaptée pour ses utilisateurs (nous, finalement) est Camel : il apporte une abstraction bienvenue à l'intégration de services mais sa syntaxe le rend difficile à utiliser.

En somme, nous pensons que les DSL sont une approche qui simplifient la tâche d'un expert dans son quotidien, mais cela doit être fait en collaboration avec lui aussi bien pour les modèles que pour la syntaxe afin d'être complètement pertinent.