Domain Specific Languages

Sensor Simulation Language | Équipe B

DSL of Things

Sommaire

Présentation

Objectif

L'objectif de ce projet, est de réaliser un langage qui permette de répondre à un besoin fonctionnel plus ou moins défini, et qui soit utilisable par quelqu'un connaissant le domaine auquel il s'applique.
Idéalement, ce langage se rapproche d'un langage naturel pour décrire le contexte et les actions à réaliser. Il doit aussi pouvoir être étendu pour intégrer de nouveaux besoins de l'utilisateur. Sans forcément être basique ou simple, il doit pouvoir être utilisé par quelqu'un qui n'a pas les bases en programmation.

Le domaine en question est celui de la simulation de capteurs IoT, dans le contexte de bâtiments intelligents. Il faut pouvoir définir des ensembles de capteurs et les relevés qu'ils génèrent, pour ensuite transmettre ces derniers à une base de donnée en vue d'une exploitation ultérieure.

Proposition

Nous avons choisi la solution MPS pour définir et implémenter ce langage. MPS permet de définir des modèles, qui représentent des concepts. Ces concepts peuvent ensuite être composés et associés les uns aux autres pour représenter un ensemble cohérent.
Un concept peut ensuite être représenté de deux manières :

- de manière textuelle pour le représenter d'une manière arbitraire;
- de manière technique, pour lui associer une (ou plusieurs en fonction du contexte) implémentation dans un langage de programmation.

En choisissant MPS, nous nous offrons la possibilité de modeler notre langage pour changer à volonté sa forme, mais nous enfermons dans une plate-forme de développement pour le moins exigeante et atypique.

Retour d'expérience

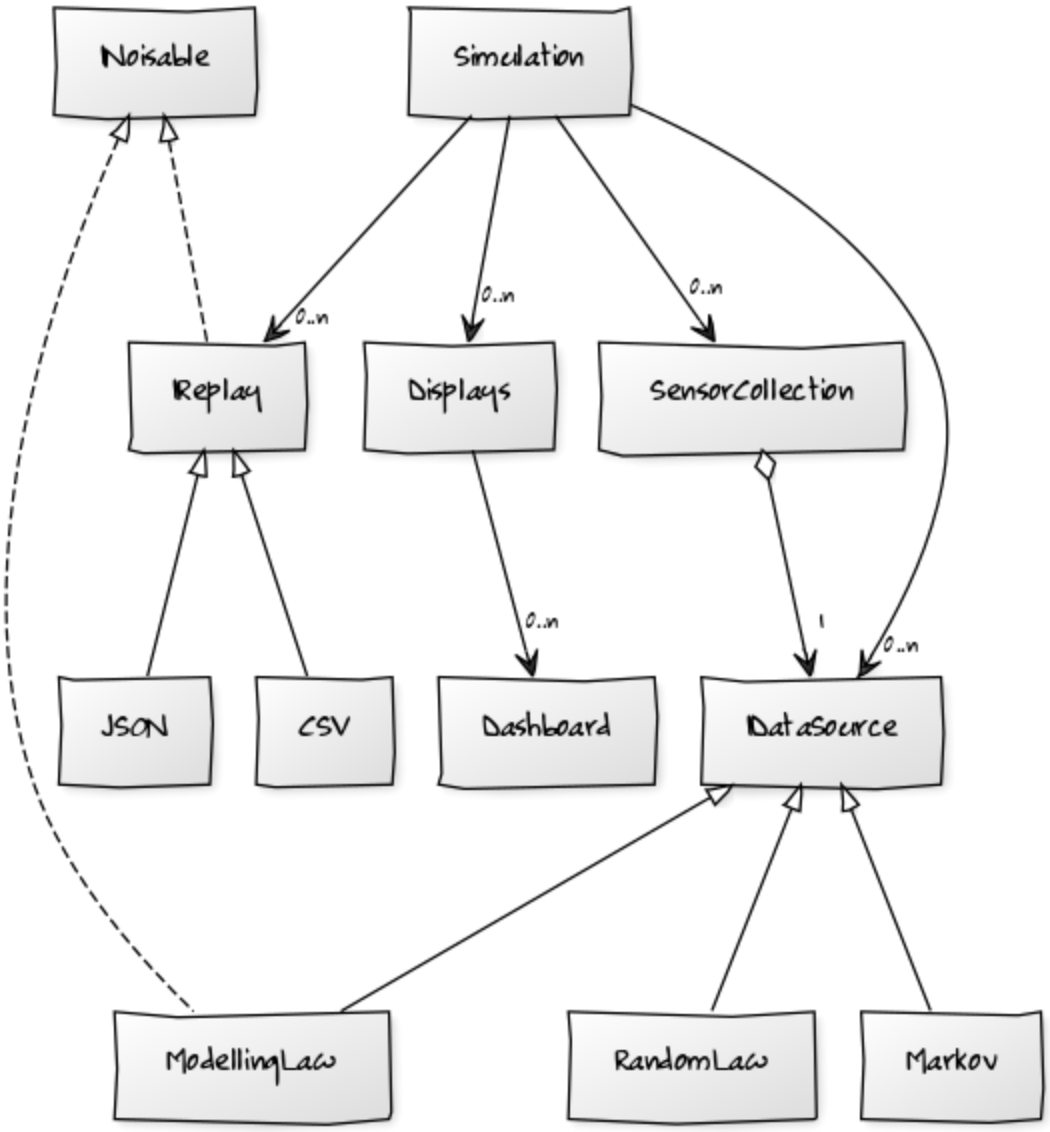
Il est indéniable que MPS est un outil très puissant et expressif. Cependant à l'image de son éditeur projectionnel, il est très dirigiste. En chaque endroit, on ne peut faire qu'un nombre limité d'actions, par rapport au contexte actuel. Cela pourrait être une bonne chose, mais la plate-forme pèche par une documentation "poussive", à la fois trop et trop peu détaillée. Et surtout, pas vraiment à jour (merci les langages dépréciés mais présents).
La configuration est aussi extrêmement fastidieuse, et ce à tout instant : que ce soit pour configurer des dépendances, pour configurer le l'IDE standalone, etc. De plus, on n'a en général que 2 ou 3 résultats pertinents lors d'une recherche Google relative au problème.
Ce projet a été sous l'aura d'une relation amour-haine avec MPS, très capable mais fatigant.

Le reste de ce rapport est constitué des éléments suivants : la présentation du modèle de notre langage, une description du langage réalisé, un détail sur l'implémentation de l'extension que nous avons choisi, un regard critique sur le langage produit et son implémentation pour finir sur une réflexion sur les DSLs dans le cadre de ce projet.

Modèle du langage

Nous nous sommes imposé une relative simplicité dans le modèle du domaine. Les trois fonctionnalités principales sont d'ailleurs mises en évidence : le rejeu de relevés, la configuration de dashboards et la simulation de grappes de capteurs ayant chacunes leurs propres sources de données.

Le modèle de l'implémentation est légèrement plus complexe car il doit correspondre aux contraintes de MPS (concepts placeholders, collections de références, etc.) et n'est pas détaillée ici car hors scope.



Description du langage et de ses fonctionnalités

La description du langage et de ses fonctionnalités est disponible dans le README présent sur le repo. Cette partie va expliquer comment un script est composé.

Création d'un script de simulation

Lorsque l'on créé une nouvelle simulation, le contenu par défaut du script est le suivant :

```
simulation TutorialSimulation
influxDB host : http://localhost:8086
grafana host  : http://localhost:3000

starts the dd-MM-yy HH:mm
ends the dd-MM-yy HH:mm

data sources:
  no data source defined !
sensor collections:
  no collection defined !
replays:
  no replay defined, but you can leave it so
grafana displays:
  no dashboards set, but it's ok that way
```

Composition du script

Le script n'est pas un simple fichier texte, mais une projection du modèle qui a été décrit plus haut. En éditant un script, on va ajouter et/ou supprimer des noeuds dans l'instance du modèle.

Le premier noeud est celui du script, qui comporte quelques propriétés (nom, url des instances, etc.) ainsi que des enfants (grappe de capteurs, sources de données, etc.). Chaque enfant est aussi un noeud, qui peut à son tour contenir des propriétés et des enfants.

En ajoutant des noeuds, on ajoute des fonctionnalités qui seront ensuite traduites en code Java pour animer la simulation.

Du fait de la nature de l'éditeur de modèle, on ne peut pas arbitrairement ajouter du texte (et donc du code) au script. On ajoute un noeud et la projection configurée lors de la création du langage est affichée dans le script.

Cela implique qu'il n'y a pas de réelle syntaxe du point de vue utilisateur : il n'y a que des noeuds projetés qui peuvent à leur tour accueillir certains types de noeuds. L'utilisateur n'a qu'à choisir quels noeuds il souhaite ajouter au script, et à remplir les champs disponibles avec les valeurs demandées.

Ce qui se rapproche le plus d'une syntaxe est la logique selon laquelle un noeud peut en accueillir un autre : il doit y avoir un lien de parenté logique. Par exemple, un ensemble de grappes de capteur peut contenir des définitions de capteurs, mais une loi de modélisation ne pourra pas accueillir de définition de dashboards Grafana : ce serait illogique.

La complétion semi-automatique affiche à l'utilisateur ce qu'il peut faire selon le contexte.

Configuration de dashboards Grafana

Contexte

Une fois les données présentes dans la base de données, il est possible de les utiliser pour afficher des graphiques sur Grafana, une webapp tierce. La configuration de cette application est fastidieuse : il faut configurer la source de données, les dashboards, les panels qui composent ces dashboards, pour finir avec les requêtes qui récupèrent les données.

L'objectif était de pouvoir configurer tout cela depuis le script créant la simulation, afin de facilement en visualiser les résultats.

Réalisation

Pour pouvoir faire cela, il faut utiliser l'API de Grafana et un client de cette API dans le langage d'implémentation de notre DSL. Malheureusement, l'API de Grafana n'est pas conçue pour réaliser toutes les opérations que nous voulions permettre d'automatiser depuis notre DSL. Il a fallu se limiter à ce qui était possible : c'est à dire la configuration de dashboards et de panels. À cause d'une limitation du client utilisé, nous ne pouvons proposer que des panels contenant des graphiques classiques. Il était hors scope d'implémenter un moyen dérobé pour réaliser ces fonctionnalités.

Une fois ces limitations et la marche à suivre identifiées, l'implémentation pour étendre le langage existant a été simple. Il s'agit de créer des concepts représentants les dashboards, de définir un éditeur (qui donne la projection textuelle d'un concept) pour chaque concept qui doit être affiché dans le script et un générateur. Ce dernier, permet de transformer les concepts en code (en Java dans ce projet) et de les intégrer dans le concept parent de simulation (voir la description du modèle).

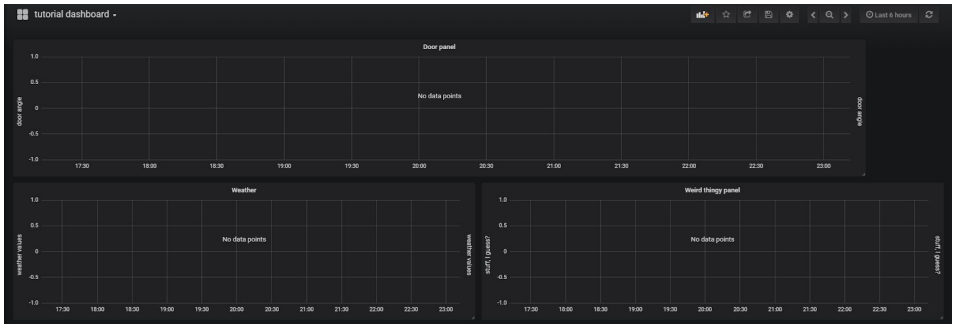
Bien que l'intégration dans la simulation soit simple, il est possible de la simplifier encore plus (voir les critiques sur l'implémentation du langage).

Résultat

Après implémentation dans le langage, le code suivant :

```
setup a dashboard named tutorial dashboard with the following rows
  panel name  : Door panel
  yAxis label : door angle
  span (on 12): 11
  -------------
  panel name  : Weather       | panel name  : Weird thingy panel
  yAxis label : weather values  yAxis label : stuff, I guess?
  span (on 12): 6               span (on 12): 6
```

donne le résultat suivant dans Grafana :



Prise de recul

Le langage

Sa “syntaxe”

L’intention qui est derrière la conception de ce DSL est d’offrir une interface textuelle proche du langage naturel à un système relativement complexe, afin que l’utilisateur final n’ai pas à avoir connaissance de sa complexité ni n’ai besoin de connaissances particulières en programmation. Avec MPS, les projections de concepts nous permettent de complètement découpler la partie *langage* et la partie *génération de code métier* du DSL.

Cela nous permet de pouvoir créer des “textes à trous” que l’utilisateur n’a qu’à remplir sans avoir à se soucier de la syntaxe ou de leur disposition, voir même de proposer des éditeurs différents pour des utilisateurs différents.

À cause de notre approche naturellement technique du problème, nous avons conçu un langage qui reprend des bases de programmation : des collections, des structures, des hiérarchies, etc. La mise en forme s’inspire d’ailleurs librement du langage de configuration YAML. Si cela rend les choses intuitives pour nous, nous n’avons aucune idée de comment ce langage serait perçu par un utilisateur non technique.

Ses fonctionnalités

Nous avons implémenté des fonctionnalités de base, permettant entre autre à l’utilisateur de configurer la simulation et les adresses des systèmes auquel le simulateur se connecte.

Nous aurions aimé en ajouter d’autres dans l’optique de rendre le DSL plus flexible pour l’utilisateur, comme une option pour configurer le format de dates à utiliser. À cause de la nature de l’éditeur, soit ces options prendraient inutilement de la place dans le script si les valeurs par défaut sont utilisées, soit seraient dissimulées derrière une abstraction, et l’utilisateur devrait avoir connaissance de leur existence et explicitement demander à les faire apparaître. Une piste pour résoudre ce problème était les *Context Assistant*s, qui permettent de faire apparaître des boutons contextuels, modifiant le modèle projeté.

Son implémentation

Généricité des concepts

Nous voulions au début utiliser des concepts parents (Interfaces) pour disposer les noeuds de configuration dans le concept de simulation. Si cela aurait permis une intégration sans modifications de ce concept lors de l’ajout de l’extension, cela aurait aussi permis d’ajouter n’importe quel noeud de configuration (collection de grappes, de sources de données, etc.) dans n’importe quel ordre dans le script de simulation.

Cela était contraire à ce que nous voulions pour l’utilisation de notre langage, et aurait nécessité une définition bien plus complexe de l’éditeur pour garantir l’ordre de présentation.

Des concepts au code métier

Pour passer d'un concept dans l'arbre du modèle à du code Java, il faut passer la phase de génération. Les concepts sont associés à des templates par des règles de réduction. Ils peuvent êtres imbriqués les uns dans les autres. L'ordre de génération est fixé par l'ordre des règles de réductions.

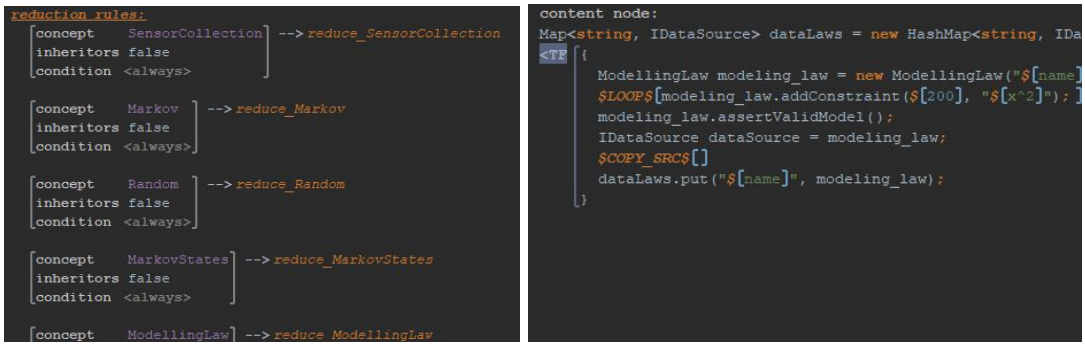


Illustration des règles de réduction et des templates (c'est joli)

Nous avons repris le principe du kernel du lab ArduinoML. Un ensemble de classe implémente le fonctionnement du système (lois, logique d'exécution, extraction de données, etc.) et une API permet d'interagir avec. Elles sont embarquées lors de la génération et le code généré y fait appel.

Le côté intéressant de cette approche est que l'on n'a qu'un seul fichier à générer, celui décrivant la configuration, et que cette solution devient réutilisable ailleurs que dans le DSL. Le point négatif est que l'on se retrouve avec un *overhead* assez important, autant (probablement) en terme d'exécution que de conception. Ne pourrait-on pas se passer du *kernel* et partitionner le code grâce aux templates par concept, afin d'optimiser finement le code produit par le DSL et plus simplement le réutiliser sur des plates-formes différentes ? Est-ce que produire une implémentation "jetable" n'aurait pas un certain intérêt dans ce contexte ?

Un DSL pour SSL, oui, mais pourquoi ?

La définition de ce DSL a très clairement répondu au besoin initial : offrir une interface textuelle utilisable par une personne non-informaticienne afin de lui faire gagner en autonomie dans son utilisation du système. Notre approche avec MPS semble cohérente avec le besoin, car l'utilisateur n'a pas à connaître de syntaxe particulière pour utiliser le produit et peut pourtant créer un programme exécutable.

Cependant, est-ce qu'un DSL sous une forme textuelle est la réponse adaptée au problème ? Nous n'en sommes pas si sûrs. Si le DSL permet une utilisation personnalisée, adaptée et efficace du système, on y gagnerait peut-être à réaliser une IHM non textuelle (car au final, notre DSL à base de blocs à trous ressemble fortement à une IHM).

Le DSL est un outil très puissant et intéressant, que l'on peut être tenté d'utiliser pour tout et n'importe quoi, mais il est parfois plus utile et pertinent de concevoir des solutions moins *over-engineered*.