UNIVERSITÉ DE NANTES

Rapport Projet Capstone MDE

Contributions à une plateforme de génération de services logiciels pour le contrôle d'objets interconnectés

Réalisé par:
TEBIB Mohammed El Amin
BELASSAL Salma
LEMCIRDI Anas
MESSAOUD Nadhir

Encadré par les membres de l'équipe *AeLoS* :
ATTIOGBÉ Jérémie Christian
ANDRÉ Pascal
ROCHETEAU Jérôme

Janvier 2018

Table des matières

Remerciements

Nous tenons à remercier nos encadrants, M. ROCHETEAU Jérôme, M. ANDRÉ Pascal et M. ATTIOGBÉ Jérémie Christian pour leur encadrement, disponibilité, leurs nombreux conseils et encouragements.

1 Introduction

Dans le cadre de notre formation de Master 2 ALMA, nous avons choisi comme sujet pour le module des projets *Capstone*, la contributions à une plateforme de génération de services logiciels pour le contrôle d'objets inter-connectés.

Nous avons pour objectif de réaliser une plateforme logicielle qui génère un ensemble de services web permettant de superviser finement des agents (objets), qui sont modélisés suivant l'approche *SAN*[1], donc nous avons un réseau de capteurs capables de prendre des mesures depuis l'environnement physique et un réseau d'actionneurs capables de modifier cet environnement, et qui communiquent à travers un système de messagerie, en respectant le protocole *MQTT*[2]. La supervision des agents intègre l'acquisition des données et le contrôle des interactions entre les agents.

1.1 Contexte

Dans notre cas d'étude, les capteurs sont liés au contrôleur, et donc ce dernier lit directement les informations nécessaires pour déclencher, en réponse, les actions correspondantes. Nous nous contentons de deux types de capteurs dans notre implémentation proposée du modèle *SAN* , il s'agit d'un capteur de *mouvements* et un deuxième pour mesurer la *luminosité* ambiante.

Notre projet porte sur la manipulation et le contrôle d'objets inter-connectés faisant sujet de l'IoT[3], ou l'internet des objets, un concept qui peut être définit comme un réseau de réseaux, qui permet grâce à des systèmes d'identifications complexes, de transmettre des données entre des objets physiques et/ou des objets virtuels. Les réseaux capteurs-actionneurs (*SAN*) constituent une nouvelle approche pour le contrôle des objets inter-connectés.
Le premier modèle de départ nous a été fourni par nos encadrants, néanmoins au fil de l'avancement du projet, ce dernier a été enrichi pour répondre aux exigences fonctionnelles de l'application qui contrôle les différents périphériques du réseau *SAN*.

Le protocole *MQTT* assure la communication entre les différents composant du système.

Le produit final étant de développer une plateforme de génération de services logiciels pour le contrôle d'objets inter-connectés, nous avons tout d'abord pris une idée sur le fonctionnement général du système à gérer.

1. Sensor and Actuator Networks
2. Message Queue Telemetry Transport
3. Internet Of Things

La composition abstraite du système à contrôler par notre plateforme est décrite dans le schéma ci-dessous :

Dans la figure 1 on peux distinguer deux grandes parties du système

1. Les intervenants externes : Ceux-ci fournissent des informations d'entrée pour le réseau SAN
   — Utilisateurs de l'application de pilotage du système : Donner des consignes, des instructions pour modifier
   — Environnement physique : La source des changements des métriques (luminosité, mouvement) et permet donc de déclencher les capteurs pour pouvoir mesurer de nouvelles valeurs d'entrée selon le type du capteur.
2. Le Réseau SAN Ce réseau comporte trois composants principaux :
   — Contrôleur : Garanti l'interaction et la coordination entre les intervenants externes et les capteurs et les actionneurs
   — Capteur : c'est un appareil qui détecte et répond à un type d'entrée provenant de l'environnement physique. L'apport spécifique peut être la lumière, la chaleur, le mouvement, l'humidité, la pression, etc. Dans notre système la sortie est transmise via $MQTT$ sur le réseau pour la lecture ou le traitement ultérieur.
   — Actionneur :

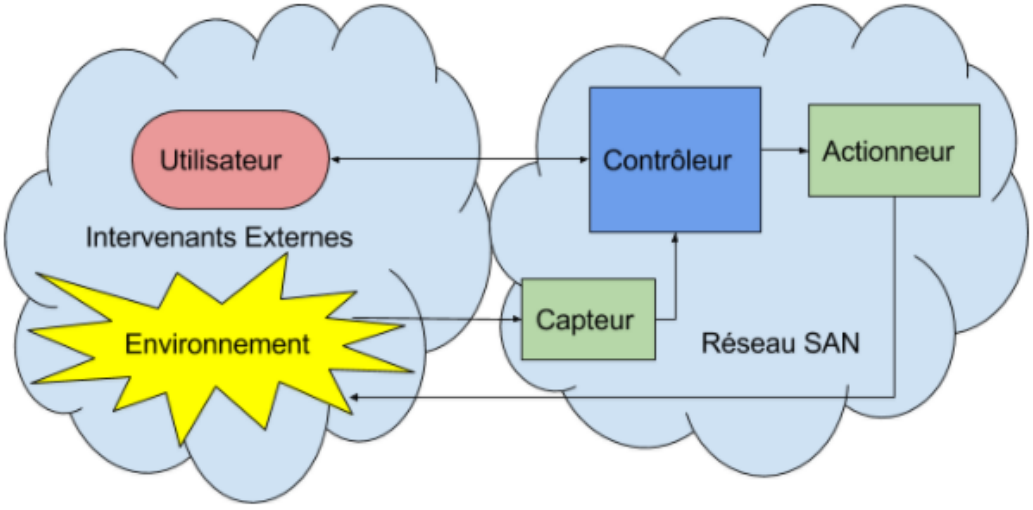

FIGURE 1: Schéma explicatif du type de réseau des capteurs-actionneurs

Le fonctionnement général du réseau est simplifié par l'algorithme ci-dessous :

1. Lorsqu'un événement est détecté par un capteur, ou qu'une commande est émise par l'utilisateur, l'information est saisie par le contrôleur
2. Activation des actionneurs concernés par le changement, et modifient si besoin, à leur tour, l'environnement physique.
3. L'environnement change, les lectures des capteurs changent (revenir à la première itération)

1.2 Motivation

Dans cette partie nous mettons en évidence la nécessité du travail demandé, la problématique et les apports des briques ajoutées à la plateforme logicielle.
Emit est l'application web qui fait l'interface avec l'utilisateur et lui donne la main pour contrôler les différentes pièces des bâtiments du campus de la Chantrerie de l'université de Nantes.
Un des objectifs considérables du projet serait la volonté d'automatiser l'ajout de fonctionnalités à Emit à partir de modèles abstraits. En effet, dans le déroulement du projet nous avons employé des techniques de développement qui se reposent sur des différents niveaux d'abstractions
(Métamodèle SAN $\longrightarrow$ Métamodèle $MQTT$ $\longrightarrow$ génération de modèles raffinés $\longrightarrow$ implémentations du code java) afin de générer par la suite les services déployables correspondants sur Emit.
La dimension importante qui motive notre projet, est de pouvoir définir un langage commun et une grammaire qui facilite l'échange entre les spécialistes du domaine métier IoT et les intervenants à la conception et le développement des services logicielles. Cette collaboration entre ces deux acteurs devrait être **perpétuelle** dans le but de garantir la richesse et surtout la cohérence .

Avec cette approche d'ingénierie dirigée par les modèles, les éventuels prochains développeurs de l'application Web de contrôle des objets connectés (Emit), pourront recevoir directement les consignes de codage, de la part d'un architecte de solutions qui aurait à son tour dessiné les modèles correspondant aux nouvelles fonctionnalités souhaitées.

— Recueillir un modèle exécutable équivalent à un modèle SAN abstrait.
— Le modèle $MQTT$ c'est un protocole de communication standard basé sur le protocole TCP/IP qui nous donne la possibilité de communiquer avec plusieurs applications SCADA [4].
— La transformation est basée sur une approche guidée par les modèles ce qui facilite la passage des modèles SAN vers $MQTT$ et rend la

transformation simple a mettre en oeuvre.

1.3 État de l'art

Le sujet du projet contient trois concepts fondamentales sur lesquelles nous avion dû rechercher et analyser l'avancement, il s'agit de l'IoT [4], la MDE [5] et les SANs [6]. C'est pour cette raison que l'état de l'art du projet est relativement laborieuse.

Nous allons commencer par suivre l'historique et les derniers impacts de l'IoT pour prendre connaissance des nouveautés du milieu dans lequel intervient.

L'IoT est né de la convergence des technologies sans fil, des systèmes micro-électromécaniques et d'Internet. En effet un objet de l'IoT, peut être à peut prêt n'importe quel périphérique connecté à internet, quelque soit son type et sa fonction.

Le concept de l'Internet des objets apparaît pour la première fois en 1995 dans le livre de Bill Gates, « The Road Ahead », mais à ce stade là c'était comme de l'encre sur du papier, dans le sens où l'utilisation même du concept a étonné la communauté internationale.

En 1998, au Massachusetts Institute of Technology (MIT), Kevin Ashton a introduit la notion d'Internet des objets. Ensuite en 1999, le MIT fonde Auto-ID. D'après l'institut : « Toute chose peut être connectée par Internet ».

A l'arrivée de l'année 2009, l'IoT a commencé à vivre un large succès, un nouvel article « L'internet des objets : le plan d'action européen » a été publié par la Commission Européenne qui présente les perspectives et les enjeux de son développement. Selon une équipe de l'ETH de Zurich, du fait des smartphones puis du nombre croissant d'objets connectés, en dix ans (2015-2025) 150 milliards d'objets devraient se connecter entre eux, avec l'Internet et avec plusieurs milliards de personnes [8]

Nous pensons que cette croissance exponentielle, est essentiellement bénéfique pour améliorer la qualité de la communication entre les objets connectés, et donc fournir plus de confort au grand public. Néanmoins nous restons nuancés là dessus, d'une part, par crainte de fortifier la dépendance à la technologie en général, et d'autre part, par abus de l'utilisation des informations personnelles et privés.

Dans notre cas de figure, nous allons explorer un exemple bien précis de l'évolution du concept de l'IoT, il s'agit de la quatrième génération de

4. Internet of Things : Internet des Objets
5. Model Driven Engineering
6. Sensor and Actuator Networks

SCADA [7] [10] qui est un système de télégestion offrant le traitement d'un grand nombre de mesures d'une installation technique, ce contrôle est caractérisé par le fait qu'il est à distance et en temps réel. SCADA a prouvé sa **scalabilité** pour gérer des réseaux complexes, d'où son emploi pour **piloter** le réseaux SAN sur lequel nous avons travaillé, est bien conseillé et justifié.

L'ingénierie dirigé par le modèle (MDE) intervient dans le niveau architectural de notre projet, parce que le besoin de passer par un modèle intermédiaire (MQTT) conforme au méta-modèle SAN du réseau s'avère cruciale. Cette nécessité est exprimé au niveau métier des web-services qui implémentent les spécifications fonctionnelles requises aux utilisateurs de la plateforme logicielle sur laquelle nous avons contribué.

1.4 Contributions

Les principales contributions techniques de ce projet comprennent 4 étapes fondamentales :

1. Dans la première étape nous avons définit un DSL (langage dédié) qui reconnaît tous les modèles *SAN* fournis par l'utilisateur en nous servant du framework Xtext [5], dont le but est de faciliter la description des modèles *SAN* en utilisant un format textuel, le résultat est un fichier source avec l'extension ".san".
2. Pour pouvoir transformer le modèle *SAN* entré en un modèle*MQTT* [9] correspondant en tirant profit du langage ATL [8] [7], le modèle source doit être sous le format xmi. Par conséquence, la deuxième étape de notre projet consiste à développer un programme java qui génère la version xmi de notre modèle avec l'extension .san.
3. Un avantage important de l'outil Xtext est de permettre la génération automatiquement d'un méta-modèle Ecore qui correspond au DSL défini au préalable, donc tous les modèles *SAN* définis par l'utilisateur seront considérés comme des instances de ce méta-modèle.
4. Comme les instances du méta-modèle générés par l'outil Xtext ne respectent pas la structure souhaitée pour pouvoir les transformer vers des modèles*MQTT* , nous avons défini dans la troisième étape une transformation ATL. Le but de cette transformation est de restructurer les modèles *SAN* .xmi et donc obtenir des versions compatibles avec notre méta modèle *SAN* écrit à la main.
   Nous nous sommes basés sur ce dernier modèle pour la rédaction de

7. Supervisory Control And Data Acquisition : système d'acquisition et de contrôle de données
8. Atlas Transformation Language

notre grammaire de règles de transformation, qui permet de fournir un modèle *MQTT* cible à partir d'un modèle *SAN* source.

5. Dans la dernière étape nous avons eu recours à deux grammaires de règles (*grammaire 1* : 4 règles, *grammaire 2* : 6 règles) dont le but est de transformer n'importe quel modèle source *SAN* vers un modèle *MQTT* .

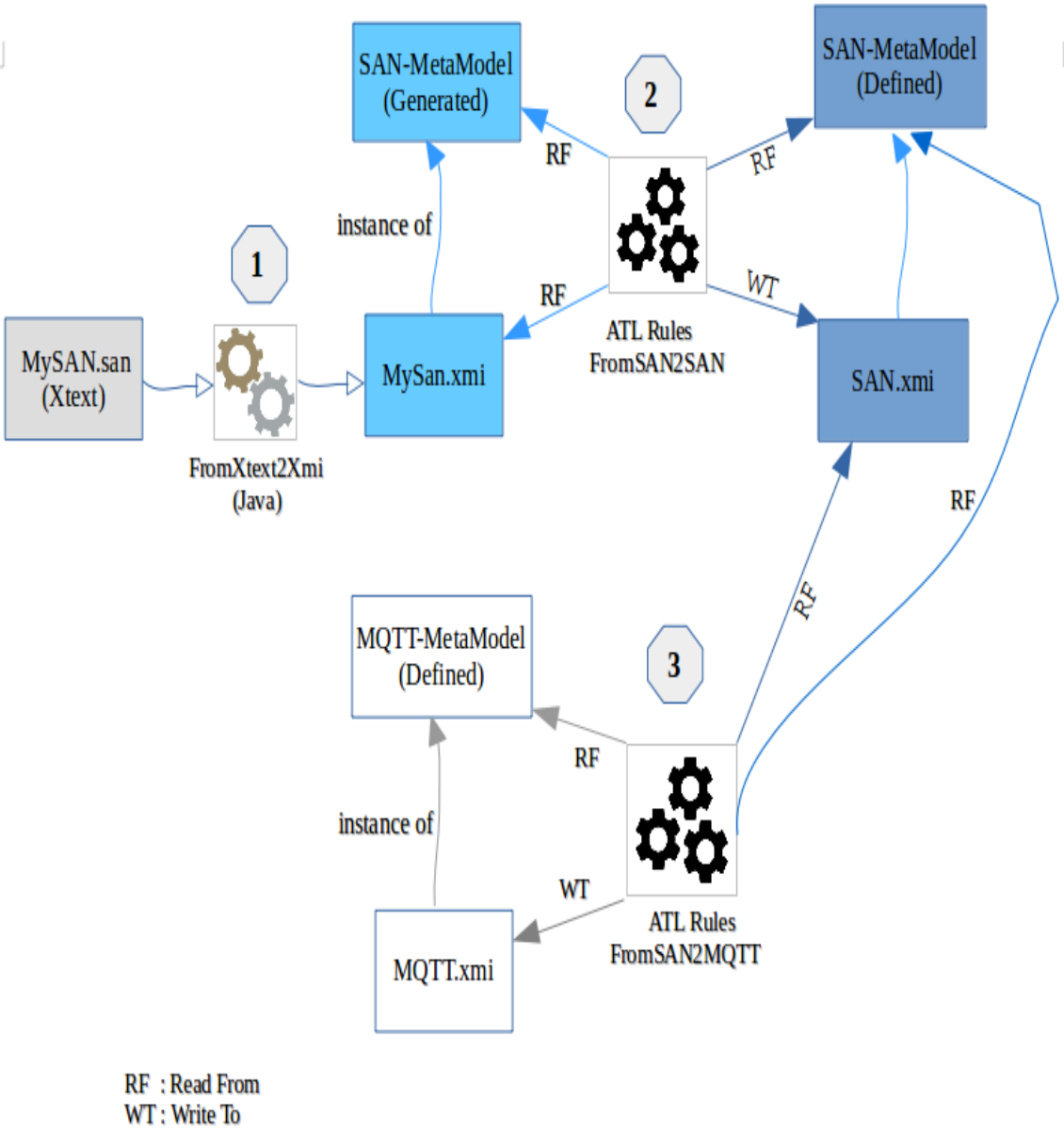

FIGURE 2: Les étapes du projet

2 Métamodèles SAN et MQTT

Un métamodèle est un modèle d'un modèle, en d'autres termes c'est un langage qui défini la structure d'un modèle, c'est pour cela que nous avons introduits ce concept pour définir deux langages pour les réseaux *SAN* et *MQTT* à l'aide du framework Ecore.

2.1 Métamodèle *SAN*

Les éléments composants le modèle *SAN* sont :

- *Network* : Il spécifie la déclaration d'un réseau *SAN* accueillant l'ensemble des éléments du modèle
- *Measurand* : Les measurands représentent les objets mesurés ou instrumentés : une pièce, un bureau, un équipement ...etc (qui peuvent être réparties géographiquement en des zones)
- *Measure* : Pour indiquer le critère à mesurer comme : luminosité, mouvement, présence ...etc avec l'unité de mesure : boolean...
- *Instrument* : Ce sont les instruments employés pour mesurer, par exemple : les capteurs de mouvement, de présence, dimmer (Dispositif électrique permettant de faire varier le flux lumineux d'un appareil d'éclairage)...
- *Parameter* : Représentent les paramètres des instruments (nom, type...etc)
- *Device* : Les appareils utilisés pour mesurer, ils ont des types selon les instruments déclarés (nom, instrument, measurand), dans les éléments qui sont mesurés (measurand) avec des '*arguments*' décrivant leur paramétrage
- Argument : Description des paramètres des instruments (broker, topic..etc) liés à un device
- *Feature* : Les caractéristiques des instruments.

une version graphique du méta-modèle se présente dans la figure 3

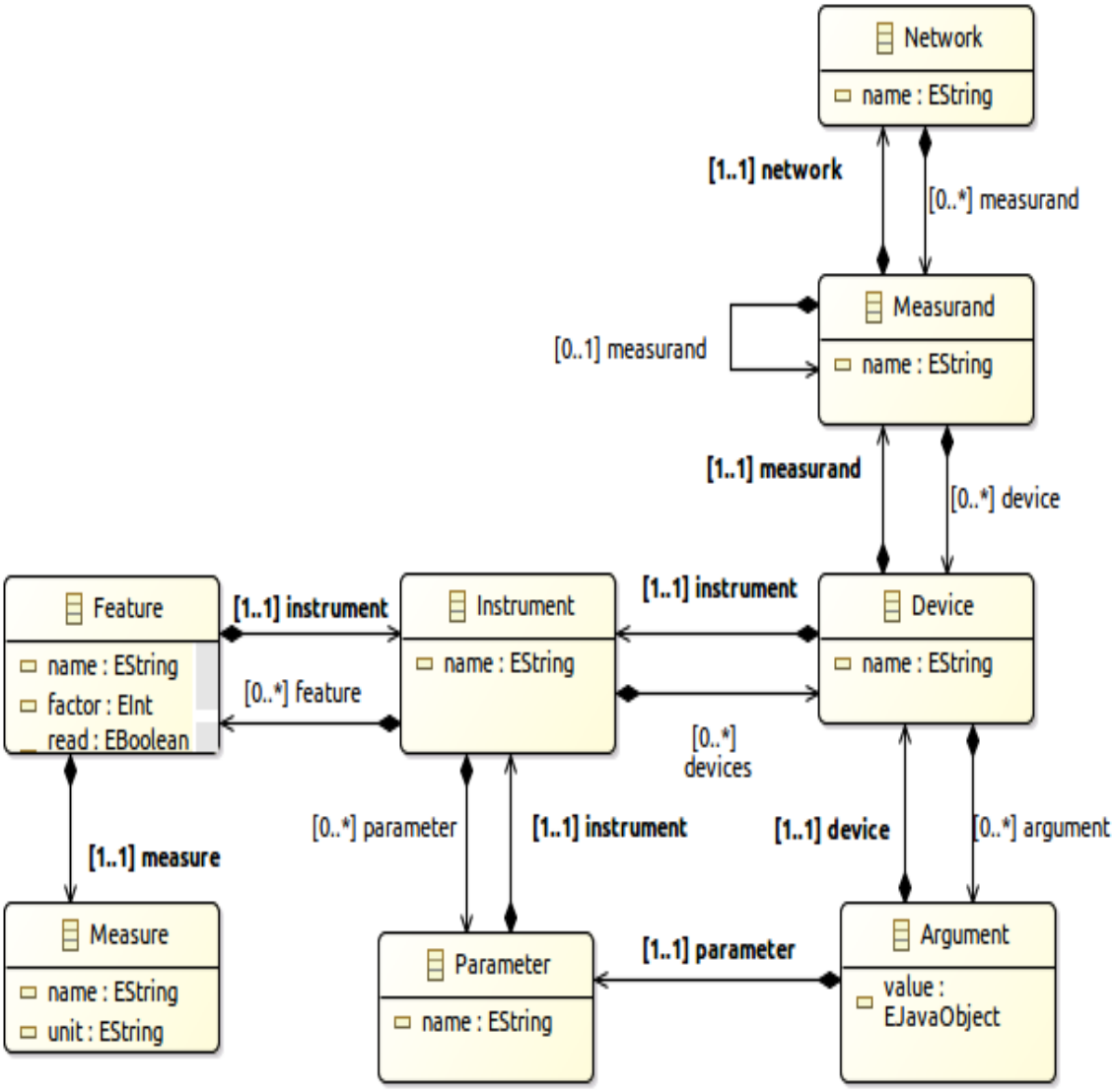

FIGURE 3: Représentation graphique du méta-modèle *SAN* avec Ecore

2.2 Méta-modèle *MQTT*

Un méta-modèle *MQTT* est composé des éléments suivants :

— *Topic* : Un topic s'occupe des messages reçus, il spécifie la destination de chaque message envoyé par un dispositif. Les messages sont représentés par des chaînes de caractères, séparés par des slashs indiquant le niveau du topic.

— *Publish/Subscribe* : en *MQTT* , chaque dispositif peut publier un mes-

sage dans un topic, en plus il peut s'inscrire à un topic pour recevoir des messages envoyés par un autre dispositif.
— *Message* : sont des paquets de données échangés entre les dispositifs (clients).
— *Broker* : tous les topics d'un protocole *MQTT* sont composés a l'intérieur d'un broker *MQTT* .Un broker reçoit tous les messages, les filtres, et enfin publie tous les messages aux clients inscrits.

Une description graphique de ce méta modèle est présentée dans la figure 4.

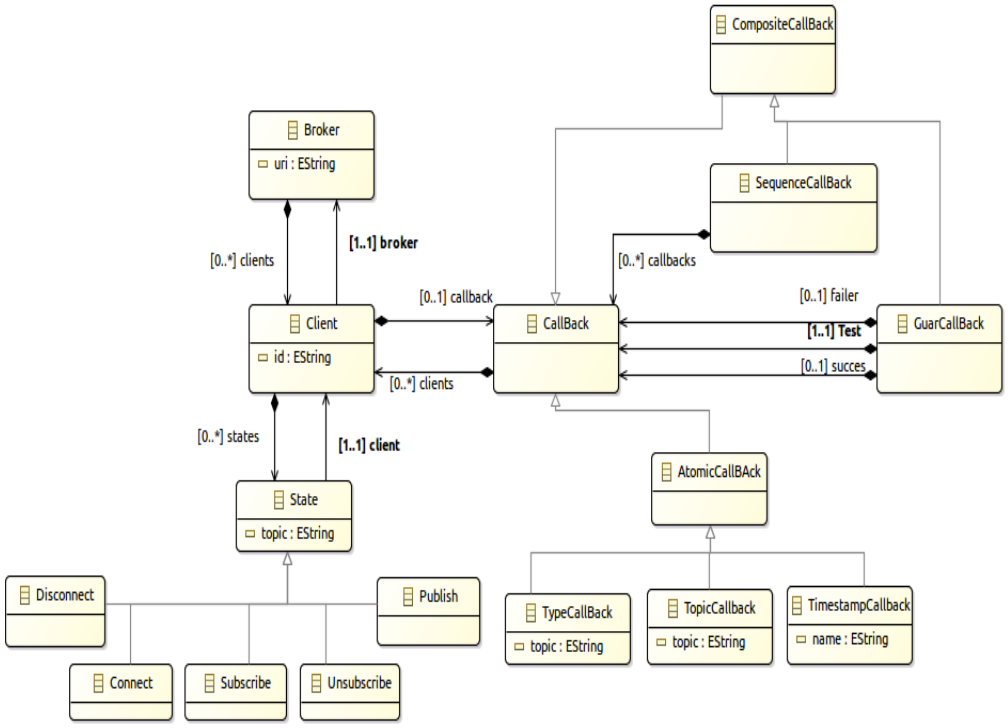

FIGURE 4: Représentation graphique du métaModèle *MQTT* en Ecore

2.3 Synthèse

Cette partie a présenté le langage SAN qui pourrait être considéré comme un standard dans le domaine de l'IoT pour représenter des objets interconnectés. De la même manière, MQTT est considéré comme un langage référence pour les communications TCP/IP. Une finalité du projet est de pouvoir définir par la suite la correspondance entre les deux langages.

3 *SAN* : Langage dédié (DSL)

Pour faciliter la description des modèles *SAN* , nous avons trouvé que la description textuelle est la mieux adaptée, et pour réaliser cela nous avons défini un DSL [9] en utilisant le framework Xtext, qui est un langage de grammaire très puissant. L'exécution de cette grammaire va nous offrir un outil bien déterminé, pour décrire des réseaux *SAN* d'une manière naturelle et flexible.

3.1 Xtext

Xtext [5] est un framework open-source pour le développement des langages de programmation et des *DSL*, il inclue un éventail complet de fonctionnalités qui sont facilement intégrées avec Éclipse tel que : Coloration syntaxique, un compilateur spécifique, et bien d'autres.

3.2 Grammaire des réseaux *SAN* en Xtext

Pour arriver à la phase de la définition de notre grammaire *SAN* en *Xtext* , nous avons d'abord commencé par une étude de l'exemple de la grammaire *SAN* fourni que vous pouvez retrouver dans l'annexe 7.2.

Nous nous sommes servi des concepts du modèle *SAN* représenté dans la section 2.1,

Ensuite, nous avons travaillé avec *Xtext* pour définir notre *DSL*, à commencer par la déclaration des propriétés de la grammaire (nom de la grammaire), puis la réutilisation des grammaires déjà existantes (*org.eclipse.xtext.common.Terminals* pour les types de base de *Java*) :

```
grammar org.xtext.example.mydsl.San
with org.eclipse.xtext.common.Terminals
```

Le mot clé '*generate*' permet la génération d'un modèle *Ecore* qui correspond à notre grammaire

```
generate san "http://www.xtext.org/example/mydsl/San"
```

L'étape d'après correspond à la définition des règles de notre grammaire, la première règle consiste à ce que notre *Model* va se constituer d'un nombre arbitraire (*) de *Type*s qui sont ajoutés (+=) à nos *instructions* :

9. Domain Specific Language

```
Model:
     instructions+=Type*;
```

Nous allons définir par la suite notre *Type* qui peut se déléguer en un de ces différentEs règles (séparé par un |) : *Network, Measure, Instrument, Parameter, Feature, Measurand, Device, Argument*

```
Type:
   Network| Measure| Instrument| Parameter|
   Feature|Measurand |Device |Argument;
```

Network est définit par le mot clé '**network**' puis le nom de notre réseau qui est une règle identifiant le *QualifiedName* par un ensemble des *ID* (voir la définition complète en *Xtext* de la grammaire *SAN* en annexe 7.1) :

```
Network:
     'network' (name=QualifiedName);
```

Measure est représenté également par son mot clé, puis une référence vers *Feature* pour définir l'unité de mesure (mot clé **of unit**) et son type (**as**) qui un type de base de Java :

```
Measure:
     'measure' (name=QualifiedName)
     ('of unit' (featurename=[Feature]|'%'))* 'as'
     javaType=PrimitivType;
```

La règle pour *Measurand*, elle est définie de la même façon que les autres, avec son mot clé mais en ajoutant une partie optionnelle (représenté par ?) de la règle qui est une référence référentielle (cross-reference), en ajoutant un autre *Measurand* optionnel dans le premier qui représente une zone :

```
Measurand:
       'measurand' name=QualifiedName ('in' measurand=[Measurand])?;
```

La règle *Feature*, fait référence à *Measure* avec le mot clé ("**as**"), puis la définition du mode cette caractéristique étant soit l'entrée (réception d'un message)"**input of**" , soit la sortie (émission d'un message)"**output of**" d'un *instrument* :

```
Feature:
     'feature' name=QualifiedName 'as' (measure=[Measure])
         ('input of' | 'output of') instrument=[Instrument];
```

Un *Parameter*, en plus de son mot clé et l'identifiant, il référence un type primitif de java avec (**as**) et l'assigne à un *Instrument* avec (**of**) déclaré :

```
Paramètre:
     'parameter' name=QualifiedName 'as' javaType=PrimitivType
      'of ' instrument=[Instrument];
```

Un *Instrument* est défini par son mot clé et son nom d'identification :

```
Instrument:
       'instrument' name=QualifiedName;
```

Device fait référence à un *Instrument* avec (**as**) qui existe (**in**) dans un *Measurand* déclaré :

```
Device:
        'device' name=QualifiedName 'as' instrument=[Instrument]
            'in' measurand=[Measurand];
```

La règle pour *Argument* définit un argument pour un paramètre '*Parameter*' avec le mot clé **for** lié à un '*Device*' déclaré après (**of**) :

```
Argument:
         'argument' name=STRING 'for' parameter=[Parameter]
         'of ' device=[Device];
```

Après la définition de toutes les règles de la grammaire *SAN* , Xtext offre la possibilité de générer un méta-modèle *Ecore* conforme à la description de cette grammaire et dont l'exécution fournit un plugin *Eclipse* généré avec *Xtext* . Ce plugin permet finalement de définir textuellement des modèles /san, à la volée, tout en respectant la grammaire *SAN* .

3.3 Parcours et reconnaissance du modèle *SAN*

Dans cette section, nous allons présenter les outils et les techniques utilisées pour réaliser le parcours d'un exemple de grammaire *SAN* .

EMF/JAVA [10]
EMF [6] est un outil open source qui fait partie des plugins Eclipse. il a été développé en 2002, pour spécifier, construire et gérer des modèles à travers une interface graphique basé sur le langage Ecore qui offre des classes permettant cela, comme : EClasse, Epackage, EAttribute ...etc, et qui fournit des outils et le support d'exécution pour les classes *Java*. Parmi ces outils, il y a *Ecore*, qui est un méta-modèle représentant le core du framework *EMF* pour décrire les modèles, ayant la capacité de persister nos modèles en utilisant la sérialisation *XMI*. Ecore permet également de tirer l'avantage d'une API réflective pour manipuler les objets génériques d'EMF.

Nous avons utilisé *EMF/JAVA* afin de pouvoir parcourir un exemple de la grammaire *SAN* qui est conforme au méta-modèle Ecore généré en se référant sur la description *Xtext* , cette manipulation nous permet d'établir une reconnaissance de nos objets générés par l'intermédiaire de l'API EMF avec leurs correspondances Java (interopérabilité d'EMF avec Java). Cette manipulation engendre le parcours intégral de l'exemple, la reconnaissance de ses différents composants, la possibilité de calculer des métriques dessus, et de pouvoir injecter du code pour rajouter des exigences ou des fonctionnalités dans le code généré des objets du modèle afin de les raffiner, une étape qui peut servir également à améliorer la génération du code.

3.4 Synthèse

Dans cette section nous avons présenté la première étape de notre contribution qui sert à définir une grammaire *SAN* flexible et maintenable, par l'exploit de l'outil *Xtext* . Maintenant étant donné que nous suivons une approche dirigée par les modèles pour réaliser notre transformation, dans la prochaine étape nous allons essayer d'expliquer brièvement les concepts de ce domaine et la manière dont nous allons implémenter le processus de transformation.

4 Transformation de modèle

Avant d'expliquer la spécification des règles de transformation, nous allons tout d'abord mettre un point rapide sur le concept de l'OMG [11] [3] qui

10. Eclipse Modeling Framework
11. Object Management Group

considère le modèle comme un élément de première classe : c'est la transformation des modèles.

4.1 Définition

La transformation des modèles est l'un des apports les plus importants de l'ingénierie dirigée par les modèles, cette technique sert à transformer un modèle source vers un modèle cible de telle sorte que chacun de ces modèles soit conforme a son méta-modèle. Ceci est élaboré par un programme composé d'un ensemble de règles de transformation qui établissent la correspondance entre les deux modèles. Dans notre projet nous avons employé dans la figure 5 cette méthodologie pour donner une définition abstraite de notre modèle de transformation.

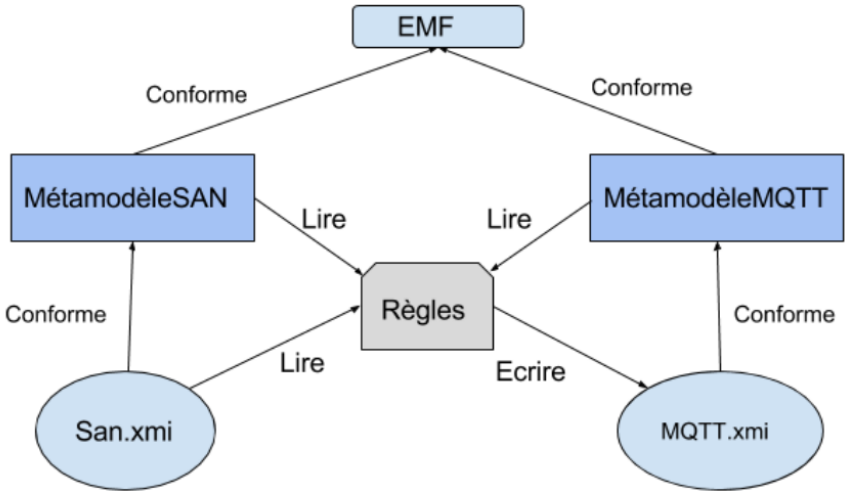

FIGURE 5: Schéma de transformation de modèle

Pour pouvoir transformer un modèle *SAN* vers un modèle *MQTT* , il faut définir ses méta-modèles en utilisant un langage de définition de mèta-modèle EMF[6]. Pour automatiser la transformation, il nous a fallu définir une grammaire de règle qui prend en entrée de lecture des modèles *SAN* , des méta-modèles *SAN* , des méta-modèles *MQTT* afin de générer comme résultat un modèle *MQTT* concret.

4.2 Choix de l'outil et la technique de transformation

Dans l'ingénierie dirigée par les modèles, il existe deux types de transformations : Modèle vers Modèle et Modèle vers Texte, de ce fait un certain

nombre d'outils spécialisées ont été proposées : Kermeta [2], ATL [7], Xtend [1], etc. Pour implémenter notre transformation nous avons choisi d'utiliser ATL parce que c'est le langage le plus approprié pour les transformations de type modèle vers modèle.

4.3 Synthèse

Dans cette section nous avons mis le point sur le concept de transformation de modèle et une vision abstraite sur la façon dont le processus de transformation adopte le concept d'un méta-modèle afin de pouvoir manipuler des modèles conformes par la suite. Ce qui nous reste maintenant c'est d'expliquer dans la prochaine partie l'ensemble des règles de transformation définies afin de fournir un modèle *MQTT* qui correspond à n'importe quel modèle *SAN* composé des éléments définies par son méta-modèle.

5 La correspondance des réseaux *SAN* en *MQTT*

Après la définition d'un modèle *SAN* en respectant le langage dédié prédéfini, et la génération de la version xmi qui correspond à ce modèle, nous allons présenter dans cette section les différentes grammaires de règles définies avec le langage ATL afin de générer comme résultat un modèle *MQTT* équivalent.

5.1 Spécification des règles de transformation

Les règles de transformation sont définies par cas à l'aide d'une fonction t(.) :

```
let n in Network
 t(n) in Broker de telle sorte que t(n).uri = n.name
   for all m in n.measurands,
     for all d in m.devices,
      for all f in d.instruments.features,
       t(f) in Client, t(f).id = d.name
       t(f).broker = t(d.measurand.network)
         exist s in Connect, s in t(f).states
           if f.read then
            exist s in Subscribe, s in t(f).states
            s.topic = m.name + / +     d.instrument.name + / + f.name
          else
```

```
exist s inPublish, s in t(f).states
s.topic = m.name + /
  +  d.instrument.name + / + f.name
```

Note : Cette spécification sera implémentée et expliquée en détail dans la partie suivante.

5.2 la transformation des *SAN* vers *MQTT* en ATL

ATL est un langage unidirectionnel (qui permet seulement de lire d'un modèle source et écrire dans un modèle cible), alors pour obtenir un modèle *MQTT* final et complet, nous avons dû lire des éléments qui se trouvent dans le modèle cible ce qui n'est pas possible en ATL. Pour résoudre ce problème nous avons choisi de définir deux grammaire de règles :
la première grammaire est composé de 3 règles :

— Règle 01 ATL

```
rule network2broker{
       from
        network : MM!Network
       to
        broker : MM1!Broker (
           uri <- network.name
        )
  }
```

Pour chaque élément dans le modèle source de type 'network' créer un 'Broker' dans modèle cible dont l'uri va prendre le nom du network.

— Règle 02 ATL

```
rule feature2mqtt1 {
  from
   feature : MM!Feature (feature.read=true)
   using {
       ins : MM!Instrument = thisModule.getInstrument(feature);
       device : MM!Device = thisModule.getDevice(ins);
       mesurand : MM!Measurand = thisModule.getMeasurand(device);
    }
 to
```

```
    client : MM1!Client(
      id <- device.name
    ),

    connect_state : MM1!Connect(
       client <- client
     ),
     subscribe_state : MM1!Subscribe(
       client <- client,
       topic <- mesurand.name+'/'+ins.name+'/'+feature.name
     )
}
```

cette règle sert a créer un 'Broker' $MQTT$ avec un client prenant l'état 'Connect' de type 'Subscriber' pour chaque feature $\in$ *instrument* $\in$ *devices* $\in$ *measurands* $\in$ *network*.

— Règles 03 ATL :

```
rule feature2mqtt2 {
   from
     feature : MM!Feature (feature.read=false)
     using {
       ins : MM!Instrument = thisModule.
       getInstrument(feature);
       device : MM!Device = thisModule.getDevice(ins);
       mesurand : MM!Measurand = thisModule.
       getMeasurand(device);
     }
   to
     client : MM1!Client(
       id <- device.name
       //ici on peut recupérer le broker du network
     ),
      connecte_state : MM1!Connect(
        client <- client
      ),
     subscribe_state : MM1!Publish(
         client <- client,
         topic <- mesurand.name+'/'+ins.name+'/'
         +feature.name
```

```
        )
    }
```

Cette règle fait la même chose que la règle 2 sauf que le client créé est de type 'Publisher'.

— Dans les règles 2 et 3 le topic est défini par « m.name + / + d.instrument.name + / + f.name » en fonction de la valeur de la propriété booléenne « read » de la Feature f.
Note : pour les règles 2 et 3, à chaque fois que le client$MQTT$ est créé, nous allons l'associer à un broker. Ce broker est le même que celui qui a été créé par la règle 1. Maintenant pour pouvoir associer ce broker au client, la première grammaire ne peut pas détecter ce broker comme c'est un élément du modèle cible. La *grammaire 2* présente une solution pour cette limitation.

La *grammaire 2* a comme entrées deux modèles sources (le modèleSAN et la modèle$MQTT$ résultant de la première transformation) :

— La première partie de cette grammaire sert à copier tous les éléments résultants de la première transformation dans le modèle cible de la deuxième transformation.

— Règle 01 ATL : pour copier les éléments de type 'Broker'

```
rule Broker2Broker {
  from
    broker1: MM1!Broker
  to
    broker2 : MM2!Broker(
      uri <- broker1.uri
    )
}
```

— Règle 2 ATL : pour copier tous les clients avec l'état 'Connect'

```
rule connect2connect{
  from
    c1: MM1!Connect
  to
    c2: MM2!Connect(
  client <- c1.client,
    topic <- c1.topic
  )
```

```
}
```

— Règle 3 ATL : pour copier tous les clients avec l'état 'Disconnect'

```
rule disconnect2disconnect{
  from
    dc1: MM1!Disconnect
  to
    dc2: MM2!Disconnect(
  client <- dc1.client
)
}
```

— Règle 4 ATL : pour copier tous les clients de type 'Publisher'

```
rule Publish2Publish{
  from
    p1: MM1!Publish
  to
    p2 : MM2!Publish(
  client <- p1.client,
      topic <- p1.topic
    )
}
```

— Règle 5 ATL : pour copier tous les clients de type 'Subscriber'

```
rule Subscribe2subscribe{
  from
    s1: MM1!Subscribe
  to
    s2: MM2!Subscribe(
      topic <- s1.topic,
  client <- s1.client
    )
}
```

— Pour finaliser le travail de transformation, il est important de compléter les règles de transformations illustrées dans la spécification, en lisant tous les éléments que nous avons pas pu reconnaître en lisant avec la *grammaire 1* (voir schéma figure 6).

— règle 6 ATL :

```
rule associate_broker{
 from
   client : MM1!Client
```

```
  using {
-- step of solution :
--1/get device from client
    device : MM!Device = thisModule.getDevice(client);
--2/get the measurand of the device
 measurand : MM!Measurand thisModule.getMeasurand(device);
--3/get the network of the device
network : MM!Network thisModule.getNetwork(measurand);
--4/get the brocker corresponds to the network
    brkr : MM2!Broker = thisModule.getBroker(network);
  }
  to
  client_with_brocker : MM2!Client(
    id <- client.id,
  broker <-  thisModule.getBroker(network)
  )
}
```

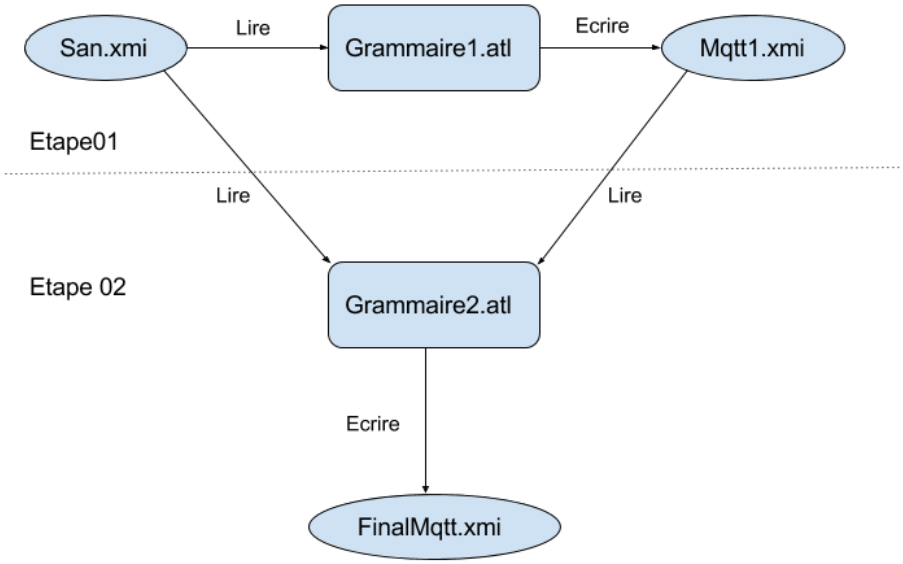

FIGURE 6: Structuration des deux grammaires

5.3 Synthèse

Dans cette partie nous avons présenté la spécification ainsi que l'implémentation des deux grammaires de règles de transformation dans lesquelles ils sont organisés comme suivant :

- Grammaire 1 : 3 règles
  - Régle 1 : network2broker,
  - Régle 2 : feature2mqtt1 (elle s'applique si l'attribut 'read' d'un 'feature' est vrai),
  - Règle 3 : (elle s'applique si l'attribut 'read' d'un 'feature' est faux).
- Grammaire 2 : 6 règles
  - les 5 premières règles permettent de copier des éléments *MQTT* résultants de la grammaire 1 dans le modèle final.
  - La règle n6 permet d'associer le 'Broker' du 'network' au client.

6 Conclusion Générale

A travers ce projet, nous avons démontré une contribution agile, pour la génération des modèles *MQTT* exécutables a travers des réseaux SAN abstraits. Nous nous sommes formés et avons manié les multiples techniques de transformation des modèles offertes par le domaine de l'ingénierie dirigée par les modèles. Cette contribution a été structurée en suivant ces étapes :

- Définir un langage dédié qui reconnaît tous les modèles *SAN* entrés par l'utilisateur avec un format textuel,
- Définir le méta-modèle des réseaux *SAN* dont le but de pouvoir construire des modèles SAN incluant les éléments suivante : network, feature, measurands, device ... etc,
- Définir le méta-modèle *MQTT* ,
- Implémenter des règles de transformation pour générer pour chaque éléments dans le modèle SAN un élément correspondant dans le modèle *MQTT* .

Une des perspectives possibles, qui peut répondre au besoin de l'utilisateur, sera d'enrichir notre transformation pour terminer la génération du modèle *MQTT* en prenant en considération tous les autres éléments restants du modèles SAN complété. ainsi que définir une transformation inverse du *MQTT* vers *SAN* . Nous proposons aussi d'utiliser le framework GMF [12] qui se repose pareillement sur la puissance de Ecore, et de créer des éditeurs graphiques pour les modèles SAN et MQTT, dans le but d'augmenter notre contribution

12. Graphical Modeling Framework

par une description graphique des modèles SAN et MQTT.
Finalement, nous regrettons de ne pas avoir le temps pour exploiter davantage les modèles MQTT produits, et ce en les implémentant au niveau du code (M0), qui est représenté par l'application EMIT.

7 Annexe

7.1 La définition de la grammaire *SAN* en *Xtext*

```
grammar org.xtext.example.mydsl.San with org.eclipse.xtext.common.Terminals

generate san "http://www.xtext.org/example/mydsl/San"

Model:
     instructions+=Type*;

Type:
      Network| Measure| Instrument| Parameter|
            Feature| Measurand |Device |Argument;

Network:
     'network' name=QualifiedName;

Measure:
     'measure' name=QualifiedName ('of unit' (featurename=[Feature]|'%'))*
         'as' javaType=PrimitivType;

Measurand:
    'measurand' name=QualifiedName ('in' measurand=[Measurand])?;

Feature:
     'feature' name=QualifiedName 'as' (measure=[Measure])
         ('input of' | 'output of') instrument=[Instrument];

Parameter:
     'parameter' name=QualifiedName 'as' javaType=PrimitivType
         'of ' instrument=[Instrument];

Instrument:
```

```
    'instrument' name=QualifiedName;

Device:
     'device' name=QualifiedName 'as' instrument=[Instrument]
        'in' measurand=[Measurand];

Argument:
     'argument' name=STRING 'for' parameter=[Parameter]
         'of ' device=[Device];


@Override
terminal ID:
      ('^')?('a'..'z'|'A'..'Z'|'_')
        ('a'..'z'|'A'..'Z'|'_'|'-'|'/'|'.'|'\"'|'0'..'9')*;

enum PrimitivType :
     integer | boolean | string | float | uri ;

QualifiedName:
       ID ('.' ID)*;
```

7.2 Exemple de modèle pour la grammaire *SAN*

```
network fr.icam.lighting
measure luminosity of unit lux as integer
measure motion as boolean
measure level of unit % as float
instrument motion-sensor
instrument luminosity-sensor
parameter broker as uri of motion-sensor
parameter topic as string of motion-sensor
parameter broker as uri of luminosity-sensor
parameter topic as string of luminosity-sensor
feature motion as motion output of motion-sensor
feature lux as luminosity output of luminosity-sensor
instrument dimmer
parameter broker as uri of dimmer
```

```
parameter topic as string of dimmer
feature level as level input of dimmer
measurand c111
measurand c115a in c111
measurand c115b in c111
measurand c113
measurand c115
device motion-sensor-c111 as motion-sensor in c111
argument 'tcp://172.21.50.3:1883' for broker of motion-sensor-c111
argument 'c111/motion' for topic of motion-sensor-c111
device motion-sensor-c113 as motion-sensor in c113
argument 'tcp://172.21.50.3:1883' for broker of motion-sensor-c113
argument 'c113/motion' for topic of motion-sensor-c113
device motion-sensor-c115a as motion-sensor in c115a
argument 'tcp://172.21.50.3:1883' for broker of motion-sensor-c115a
argument 'c115/1/motion/' for topic of motion-sensor-c115a
device motion-sensor-c115b as motion-sensor in c115b
argument 'tcp://172.21.50.3:1883' for broker of motion-sensor-c115a
argument 'c115/2/motion/' for topic of motion-sensor-c115b
device luminosity-sensor-c111 as luminosity-sensor in c111
argument 'tcp://172.21.50.3:1883' for broker of luminosity-sensor-c111
argument 'c111/luminosity' for topic of luminosity-sensor-c111
device luminosity-sensor-c113 as luminosity-sensor in c113
argument 'tcp://172.21.50.3:1883' for broker of luminosity-sensor-c113
argument 'c113/luminosity' for topic of luminosity-sensor-c113
device luminosity-sensor-c115a as luminosity-sensor in c115a
argument 'tcp://172.21.50.3:1883' for broker of luminosity-sensor-c115a
argument 'c115/1/luminosity' for topic of luminosity-sensor-c115a
device luminosity-sensor-c115b as luminosity-sensor in c115b
argument 'tcp://172.21.50.3:1883' for broker of luminosity-sensor-c115b
argument 'c115/1/luminosity' for topic of luminosity-sensor-c115b
device dimmer-c111 as dimmer in c111
argument 'tcp://172.21.50.3:1883' for broker of dimmer-c111
argument 'c111/dimmer' for topic of luminosity-sensor-c111
device dimmer-c113 as dimmer in c113
argument 'tcp://172.21.50.3:1883' for broker of dimmer-c113
argument 'c113/dimmer' for topic of luminosity-sensor-c113
device dimmer-c115 as dimmer in c115
argument 'tcp://172.21.50.3:1883' for broker of dimmer-c115
argument 'c115/dimmer' for topic of luminosity-sensor-c115a
```

Table des figures

Références

[1] http ://www.eclipse.org/xtend/.

[2] http ://www.kermeta.org.

[3] http ://www.omg.org/.

[4] Stuart A Boyer. *SCADA : supervisory control and data acquisition*. International Society of Automation, 2009.

[5] Sven Efftinge and Markus Völter. oaw xtext : A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.

[6] Eclipse Foundation. Mqtt for sensor networks (mqtt-sn) protocol specification.

[7] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl : a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[8] Dirk Helbing Evangelos Pournaras. Share/bookmark society : Build digital democracy , nature, 2015-11-02. 2015.

[9] Andy Stanford-Clark and Hong Linh Truong. Mqtt for sensor networks (mqtt-sn) protocol specification. *International business machines (IBM) Corporation version*, 1, 2013.

[10] University of Tirgu-Mures Traian Turc. Scada systems management based on web services. 2015.