

Instanciation d’une “Smart Home” en OWL et recherche des ses propriétés en SPARQL

TRAVAIL DE BACHELOR

SAMUEL YERLY

Mai 2014

Superviseurs du travail :

PROF. DR. JACQUES PASQUIER–ROCHA & ALBRESHNE ABDALADHEM
Software Engineering Group

“Productivity is never an accident. It is always the result of a commitment to excellence, intelligent planning, and focused effort.”

- *Paul J. Meyer*

Résumé

De nos jours, les systèmes intelligents résidentiels sont de plus en plus nombreux afin d'améliorer la vie quotidienne de la population et la recherche sur ceux-ci est intéressante. Dans ces systèmes intelligents, des objets connectés sont utilisés avec des actuateurs et des senseurs dans divers scénarios comme l'économie d'énergie. L'intégration de ces objets dans des applications n'est pas aisée et plusieurs solutions sont testées pour trouver une solution afin de combiner des services et des composants intelligents pour qu'ils fonctionnent ensemble.

Le groupe Génie Logiciel de l'Université de Fribourg propose d'utiliser une ontologie de domaine pour modéliser un environnement intelligent afin de découvrir et d'utiliser les services de celui-ci. L'ontologie est la partie principale du framework proposé car elle décrit les différents concepts de l'environnement, les relations entre ceux-ci et les propriétés.

Préambule

Avant-propos

Etudiant en informatique à l'Université de Fribourg, j'ai réalisé mon travail de Bachelor au sein du groupe de recherche Génie Logiciel dirigé par le Professeur Dr. Jacques Pasquier. Ce groupe de recherche fait partie de la faculté des Sciences de l'Université de Fribourg en Suisse.

C'est le doctorant Albreshne Abdaladhem, membre de ce groupe de recherche, qui m'a suivi durant tout ce travail.

Remerciements

Je tiens en tout premier à remercier Mr Albreshne Abdaladhem qui m'a suivi durant tout mon travail. Je lui suis reconnaissant pour son aide et sa grande disponibilité. Je voudrais aussi remercier Mr Jacques Pasquier, chef du groupe Génie Logiciel, d'avoir proposé ce sujet qui m'a permis d'acquérir beaucoup de nouvelles connaissances.

Notations et conventions

Plusieurs termes vont être présentés durant ce travail. Les classes seront représentées en écriture normale et commenceront par une majuscule : Classe ou UneClasse. Les propriétés seront écrites en italique: *propriété* ou *unePropriété* et les instances en écriture normale en minuscule : instance ou uneInstance.

Ces concepts seront développés afin d'expliquer leur définition et leur rôle dans ce projet.

Table des matières

1 Introduction	10
2 Ontologies : Définition et raison de leur utilisation	12
2.1 Définitions	12
2.2 Conceptualisation de l'information	13
2.3 Représentation des données	14
2.4 Modélisation de données	15
3 Ontologies : OWL et Protégé	16
3.1 Introduction aux ontologies	16
3.2 Les classes	17
3.3 Les instances	19
3.4 Les propriétés	19
3.5 Création d'une ontologie	23
3.5.1 Création des classes	23
3.5.2 Création des propriétés	28
3.5.3 Création des restrictions	32
3.5.4 Création des propriétés de données	38
3.5.5 Création des instances	41
3.5.6 La consistance d'une ontologie	46
4 SPARQL	48
4.1 Définition	48
4.2 SPARQL dans Protégé : exemples de requêtes	51
5 La Smart Home	55
5.1 Présentation du projet et du scénario	55
5.2 L'ontologie de la Smart Home	57
5.2.1 Les classes	58
5.2.2 Les propriétés d'objets	66
5.2.3 Les restrictions	68
5.2.4 Les propriétés de données	70
5.2.5 L'instanciation de la maison	73
5.3 Le framework	76
5.3.1 L'architecture du framwork	76

5.3.2 Un scénario.....	77
5.3.3 L’instanciation de l’ontologie	79
5.3.4 Le registre des services.....	81
5.3.5 Le processus de découverte : l’utilisation des objets intelligents.....	86
6 Conclusion	90
Références	92

Liste des images

Figure 1 Les onglets dans Protégé.....	17
Figure 2: Illustration d'une propriété transitive (pris dans [6])	21
Figure 3 : Illustration d'une propriété symétrique (pris dans [6]).....	21
Figure 4 : Illustration d'une propriété asymétrique (pris dans [6])	22
Figure 5 : Représentation imagée d'une propriété réflexive (Source [21]).....	22
Figure 6 : Illustration d'une propriété irreflexive (pris dans [6])	23
Figure 7 : L'onglet Classes dans Protégé	23
Figure 8 : Les trois boutons pour gérer les classes dans Protégé	24
Figure 9 : Hiérarchie de la classe SmartphoneComposants sous Protégé.....	24
Figure 10 : Les différentes actions possibles pour une classe disjointe dans Protégé	25
Figure 11 : Affichage des classes disjointes dans Protégé	25
Figure 12 : Les sous-classes d'AppareilphotoComposant	26
Figure 13 : Les sous-classes de MemoireComposant	26
Figure 14 : Hiérarchie complète de la classe SmartphoneComposants.....	27
Figure 15 : Hiérarchie de la classe SmartphoneProtection.....	27
Figure 16 : Onglet Properties dans Protégé.....	28
Figure 17 : Créer ou définir une propriété d'objet inverse dans Protégé	29
Figure 18 : Hiérarchie des propriétés d'objets pour l'ontologie Smartphone	30
Figure 19 : Définir une propriété d'objet comme étant fonctionnelle.....	31
Figure 20 : Le domaine et le range de la propriété <i>aBatterie</i>	32
Figure 21 : Panel pour créer une restriction	33
Figure 22 : Création d'une nouvelle expression dans Protégé	34
Figure 23 : Création d'une restriction	35
Figure 24 : Affichage d'une restriction existentielle.....	35
Figure 25 : Affichage d'une restriction universelle.....	36
Figure 26: Affichage d'une seconde restriction universelle.....	36
Figure 27 : Affichage d'une restriction de cardinalité	38
Figure 28 : Onglet pour les propriétés de données dans Protégé	38
Figure 29 : Les propriétés de données pour définir les marques de chaque composant	39

Figure 30 : Choix du domaine et du range pour une propriété de données.....	39
Figure 31 : Création de la propriété de données « couleur ».....	40
Figure 32 : Choix du domaine et du range pour la propriété couleur.	40
Figure 33 : Onglet pour créer des instances dans Protégé.....	41
Figure 34 : Protégé affiche les restrictions non respectées dans un encadré rouge.....	42
Figure 35 : Création de'une instance de DouzeMPAppareilphoto liée à une instance de Smartphone.....	42
Figure 36 : Sélection de la marque d'un appareil photo.....	43
Figure 37 : Création des instances pour les trois composants.....	43
Figure 38 : Création d'une instance de DataSmartphone et de ses composants	44
Figure 39: Extrait de la représentation graphique de l'ontologie SmartPhone	45
Figure 40 : Vérifier la consistance d'une ontologie grâce au reasoner	46
Figure 41 : Ouvrir le panel SPARQL dans Protégé	51
Figure 42 : Déclaration des espaces de noms.....	52
Figure 43 : Affichage d'un résultat d'une requête SPARQL dans Protégé.....	52
Figure 44 : Affichage de toutes les propriétés de données de mon ontologie.....	52
Figure 45 : Affichage de toutes les instances de Smartphone et leur(s) composant(s)	53
Figure 46 : Schéma de la maison intelligente (pris dans [11]).....	56
Figure 47 : Représentation schématique de l'ontologie de la smart home (pris dans [11])	57
Figure 48 : Les classes de l'ontologie de la smart home.....	58
Figure 49 : Hiérarchie des classes pour les objets.....	60
Figure 50 : Hiérarchie des classes pour les acteurs	61
Figure 51 : Hiérarchie des classes pour les senseurs.....	62
Figure 52 : Hiérarchie des classes pour les publishers.....	63
Figure 53 : Hiérarchie des classes pour les actions	64
Figure 54 : Hiérarchie des classes pour les requêtes	65
Figure 55: Hiérarchie des classes pour les événements	66
Figure 56 : Les propriétés d'objets de la smart home	67
Figure 57 : Restrictions pour les instances de la classe Lamp	68
Figure 58 : Restriction pour les instances de la classe LampController.....	69
Figure 59 : Restrictions pour les instances de la classe DayLightSensor	69
Figure 60 : Restriction pour les instances de la classe LightChangingEventPublisher	69
Figure 61 : Les propriétés de données pour la smart home.....	71
Figure 62 : Spécification du domaine et du range de la propriété brightness	72

Figure 63 : Restriction pour les instances de la classe Location	73
Figure 64 : Création d'une instance d'une pièce	74
Figure 65 : Création d'une instance d'AirConditioner	75
Figure 66 : WSDL et OWL-S locations pour l'action switchOf_18	76
Figure 67: La vue d'ensemble de l'architecture du framework (pris dans [15])	77
Figure 68: La maison intelligente du scénario présenté (pris dans [15])	78
Figure 69: Registre des services, Discovery et tous les processus appelés (pris dans [15])	79
Figure 70: Code extrait d'une instanciation d'une lampe intelligente et de ses services (pris dans [15])	80
Figure 71: L'ontologie de domaine simplifiée et son instanciation pour un maison spécifique (pris dans [15])	81
Figure 72: Message SOAP envoyé par un actuateur LampController (pris dans [15])	82
Figure 73: Représentation de l'instanciation de la maison intelligente sous la forme d'un arbre	86
Figure 74: Extrait de code GPL4SE et la requête SPARQL correspondante (pris dans [15]).	87
Figure 75: L'interface de la maison intelligente pour contrôler les invocations de services....	89

1

Introduction

De nos jours, beaucoup de recherches sont faites dans les domaines des environnements intelligents. Il est intéressant de connaître les moyens et les technologies utilisées pour créer et contrôler un environnement intelligent. En effet, il y a deux problèmes à résoudre. Le premier est la création et la description de l'environnement. Le second est le contrôle de celui-ci et la communication entre les différents services proposés.

Dans ce travail, je présente la solution que propose le groupe Génie Logiciel de l'Université de Fribourg. Le groupe a défini une architecture logicielle pour gérer un environnement intelligent qui se compose de plusieurs parties qui seront expliquées dans le chapitre 5. Il utilise entre autre une ontologie comme base de données et des Web Services pour contrôler cet environnement et communiquer dans celui-ci. Il est présenté dans le cinquième et dernier chapitre de ce travail. Le groupe s'est basé sur l'idée que l'environnement intelligent peut être une maison intelligente qui possède différents objets intelligents. Ces objets sont associés à des actuateurs ou/et à des senseurs qui fournissent des services via des Web Services. Il est vrai qu'actuellement, il est déjà possible de contrôler sa maison grâce à de nombreux systèmes déjà en place et de nombreux projets naissent chaque mois. Le but du groupe n'est pas de proposer une application ou un logiciel pour contrôler une maison intelligente, mais c'est de proposer une solution technologique afin de créer et contrôler cet environnement.

Le groupe propose d'abord de décrire l'environnement de manière syntaxique mais aussi sémantique. Pour cela, l'utilisation d'un langage du Web sémantique est appropriée. Le Web sémantique propose des langages conçus pour les données du Web comme RDF, OWL et XML. Ces langages ont l'avantage qu'ils permettent de décrire des choses et les liens qui les unissent. RDF et OWL vont être utilisés durant ce travail.

Comme mentionné auparavant, le groupe propose une ontologie pour décrire l'environnement. J'ai donc utilisé OWL, qui est basé sur RDF, afin de définir une ontologie Web structurée. L'ontologie va permettre de représenter un environnement hiérarchisé, une maison intelligente dans ce cas, et donc les objets et leurs différentes relations qui le constituent.

Pour le second problème qui est le contrôle de l'environnement et donc de cette maison intelligente, les Web Services ont été utilisés dans le framework. Chaque Web Service correspond à un service dans l'ontologie et sont décrits grâce au langage WSDL. Le framework a également besoin de lire l'ontologie afin de la connaître et de proposer les différents Web Services. Le langage SPARQL est utilisé pour effectuer différentes requêtes sur les données web et donc sur l'ontologie.

Durant ce travail, j'ai d'abord du étudier et utiliser les différentes notions tel que l'ontologie, l'OWL, le RDF et SPARQL qui étaient toutes nouvelles pour moi. J'ai donc commencé par créer l'ontologie de la maison intelligente, puis créer des requêtes en SPARQL utiles pour le framework afin de découvrir l'ontologie avant d'apprendre à utiliser le framework pensé par le groupe.

J'ai choisi ce sujet pour mon travail de Bachelor car j'adore acquérir de nouvelles connaissances surtout dans le domaine du Web qui m'intéresse plus particulièrement dans le vaste monde de l'informatique et parce que le sujet est d'actualité. Ce travail est intéressant car il permet de toucher à plusieurs technologies, d'apprendre des nouvelles façons de penser et pouvoir comparer avec mes connaissances actuelles.

2

Ontologies : Définition et raison de leur utilisation

2.1 Définitions

Pour ce travail, le choix s'est porté sur une ontologie pour stocker les données et définir les concepts de notre "Smart Home" dite "Maison intelligente". En effet, elle sera composée, comme dans la réalité, de plein d'objets qui pourront être intelligents ou non. Ceux qui nous intéressent sont bien évidemment ceux qui le sont. Un objet intelligent est un objet qui peut communiquer en nous fournissant des informations ou/et effectuer une action.

Pour commencer, il est essentiel de définir l'ontologie. L'ontologie est la base de ce travail et il est primordial de bien comprendre ce que c'est, son rôle et pourquoi on l'utilise. Une ontologie regroupe un ensemble de concepts qui permettent de représenter explicitement le sens d'un champ d'informations, la sémantique d'un domaine. Une ontologie constitue donc un modèle de données qui représente un ensemble de concepts ainsi que les relations entre ceux-ci. Mon ontologie va donc contenir différents concepts de la maison intelligente ainsi que leurs liens que je détaillerai un peu plus tard.

Le concept d'ontologie a été défini en premier par des philosophes afin de décrire la réalité au début des années 1990. «Le terme ontologie est entré dans l'informatique par un sous-domaine de l'intelligence artificielle symbolique qui s'intéresse à la représentation des connaissances»¹. Plusieurs autres disciplines se sont aussi intéressées à la modélisation des informations telles que l'informatique l'a redéfini. L'une des visions les plus simples de ce concept dans l'informatique est proposé par Agarwal (2005) [1] qui stipule que « l'ontologie

est donc la manifestation d'une compréhension commune d'un domaine qui est convenu entre un certain nombre d'agents et qu'un tel accord facilite des communications fiables et efficaces de sens, qui entraîne à son tour d'autres avantages tels que l'interopérabilité, la réutilisation et le partage. »

Contrairement aux bases de données qui comme le nom l'indique se base sur les données, l'ontologie se base sur des concepts et leurs liens. On va donc pouvoir définir une hiérarchie en créant des classes, des propriétés et des entités ainsi que des relations entre celles-ci. Chaque classe est un ensemble d'entités et utilise des propriétés pour créer des contraintes.

On pourrait facilement trouver des relations directes entre une ontologie et une base de données. Une classe pourrait correspondre à une table relationnelle, une propriété à un attribut relationnel, etc... Toutefois, cette comparaison n'est pas aussi triviale que cela puisse sembler. Cette comparaison va être détaillée dans les prochains chapitres.

2.2 Conceptualisation de l'information

Un schéma de base de données fournit une représentation qui est conçue principalement pour répondre aux exigences d'une application bien définie de sorte que lorsque ses besoins changent, le point de vue et le schéma doivent également être modifiés.

On remarque souvent ce cas dans certaines sociétés qui veulent modifier le schéma et les concepts de leur organisation. Cela pose beaucoup de problèmes car on doit souvent repartir de zéro. Modifier une base de donnée peut être très complexe dans certains cas, cela prend du temps et ça coûte cher.

Ce genre de problème est beaucoup moins récurrent avec les ontologies. En effet, une fois les concepts et les liens définis, on peut l'utiliser comme base dans plein de projets et l'étendre comme on le souhaite. Le fait qu'une ontologie possède un couplage plus faible qu'une base de donnée est aussi un avantage dans son utilisation. Une ontologie permet donc une plus forte indépendance fonctionnelle. En effet, on ne peut prendre que certains composants, certaines classes, certains liens par exemple selon ce que l'on veut produire. L'ontologie a donc un fort avantage au niveau de son interopérabilité et de sa réutilisabilité. L'héritage et les contraintes sont aussi un point fort de l'ontologie. Toutes les sous-classes possèdent les mêmes restrictions et les mêmes propriétés que la superclasse grâce à l'héritage.

Les bases de données sont de plus en plus utilisées actuellement sur les sites web et les applications web, mais elles possèdent comme dit ci-dessus des désavantages comparé à d'autre technologies, surtout dans le contexte du Web sémantique qui vise l'échange de données par des méthodes communes. Selon le W3C, « le Web sémantique fournit un modèle qui permet aux données d'être partagées et réutilisées entre plusieurs applications, entreprises et groupes d'utilisateurs ». [2] C'est exactement le but de ce projet. En effet, notre maison intelligente aura des objets qui communiqueront entre eux et avec des utilisateurs via des données. On veut construire ce concept pour qu'il puisse être réutilisé comme expliqué précédemment. L'ontologie est au cœur des modèles du Web sémantique.

Les bases de données présentent plusieurs problèmes dans un contexte du Web sémantique, car l'information ne peut pas être sémantiquement annoté. D'une part, on a besoin d'utiliser une requête pour afficher le contenu des bases de données et la description sémantique de celle-ci est représentée par son schéma, souvent indisponible ou même inutile car il ne peut pas être exploitée en fonction du format choisi pour le représenter, ce qui n'est pas le cas avec les ontologies et le langage OWL qui seront introduits plus tard.

2.3 Représentation des données

Contrairement aux bases de données, il n'y a pas de règles pour définir une instance d'une ontologie. En effet, il n'y a aucune contrainte si on souhaite créer une nouvelle instance de l'ontologie. Une base de donnée doit assurer l'intégrité des données à tout moment. Par exemple, un tuple ne peut pas être inclus si toutes les contraintes sémantiques du schéma ne sont respectées (clé primaires, clé étrangères, ...). Dans une ontologie, il est possible de l'ajouter même si les règles ne sont pas respectées, mais l'intégrité n'est pas garantie à tout moment.

Il y a aussi des contraintes à respecter dans une ontologie sinon cela n'aurait aucun sens. On appelle cela l'intégrité des contraintes. On verra par la suite qu'on peut utiliser un "reasoner" pour résoudre ce genre de problèmes et garantir la consistance d'une ontologie dans l'outil que j'ai utilisé pour créer une ontologie. Ce reasoner va nous informer sur ce qui appartient à l'ontologie et ce qui suit donc les contraintes et ce qui ne les suit pas! Il a aussi un second rôle très pratique. En effet, il va se servir de la hiérarchie et des contraintes déjà établies pour

extraire de nouvelles informations. Il va donc plus loin qu'une simple requête classique. Son utilisation sera détaillée plus tard.

2.4 Modélisation de données

Les ontologies ont tout de même leur point faible. Leur sémantique dépend de la langue de représentation. Il existe plusieurs langages spécifiques pour représenter une ontologie: KIF; RDF, OWL, ... [1] Il existe aussi plusieurs outils qui permettent de générer le code via une interface: Protege, WebOde entre autres. Des pertes sémantiques ou des incompatibilités peuvent se produire si on veut faire une conversion entre ces différentes représentations. Cela reste un inconvénient gérable, car OWL [3] devient la langue de l'ontologie la plus étendue grâce au Web sémantique. OWL est considéré aujourd'hui comme la norme de celui-ci.

Dans ce travail, la norme OWL qui se base sur RDF [4] (Resource Description Framework) et l'outil Protégé pour concevoir l'ontologie de la Smart Home sont utilisés. « RDF est la première brique des standards du Web sémantique. » [5] RDF permet de décrire des ressources du Web et utilise les URI afin de les identifier et décrire les relations entre celles-ci. Tout ce qui peut être représenté par un URI sur le Web est considéré comme une ressource que ça soit une page Web, une image, un document, etc... RDF est structuré grâce à des triplets. Ils décrivent et relient des ressources comme je vous l'expliquerai dans le chapitre 4.

3

Ontologies : OWL et Protégé

3.1 Introduction aux ontologies

OWL ou Ontology Web Language est un langage qui permet de décrire une ontologie structurée. C'est un schéma lourd contrairement à RDF qui permet une description plus précise. OWL permet aux outils qui l'utilisent (par exemple Protégé) de lire les données, de les vérifier grâce aux contraintes, de déduire de nouvelles connaissances grâce à un reasoner et d'extraire aussi certaines informations. La version actuelle et celle utilisée dans ce travail est OWL 2.0 qui permet de méta-modéliser, c'est-à-dire créer des concepts de concepts. La définition d'une ontologie possède un ensemble de définitions de classe, de propriété, de méta-description et d'individus.

Durant ce travail, l'outil Protégé est utilisé afin de modéliser les différents concepts et leurs liens. Il a été décidé avec Albreshne Abdaladhem d'utiliser une version antérieure, la version 3.4.7, car la création automatique de nouvelles instances est plus rapide et fonctionnelle que dans la version la plus récente. Protégé va donc générer à notre place du code OWL avec une syntaxe RDF/XML grâce à une interface simple.

Même si Protégé génère automatiquement tout le code, il est nécessaire tout de même de comprendre ce qu'il fait. En effet, plus tard lorsque des requêtes seront effectuées sur l'ontologie afin d'extraire des informations et de les utiliser dans un framework, des propriétés d'OWL devront être utilisées dans les requêtes.

Comme déjà mentionné une ontologie OWL consiste en un ensemble d'instances nommées "Individuals", de propriétés et de classes. OWL est destiné à la description de classes grâce aux caractéristiques des propriétés. Dans Protégé, on retrouve exactement ces 3 onglets comme on peut le constater sur la figure 1.



Figure 1 Les onglets dans Protégé

Pour présenter Protégé, son fonctionnement et les différents concepts qu'il propose, je me suis inspiré du tutoriel de l'Université de Manchester [6] que j'ai moi-même suivi au tout début de mon travail avant de créer l'ontologie pour la Smart Home.

Durant cet apprentissage de Protégé, j'ai créé une nouvelle petite ontologie afin d'expliquer toutes les bases nécessaires à la création d'une ontologie pour un projet tel que la Smart Home. Le but est de créer une ontologie qui permettrait aux gens de créer leur propre téléphone portable évolutif en choisissant les composants qu'ils veulent et leurs caractéristiques. Cette idée m'est venue du projet Ara [7] de Google, initialement pensé par Motorola. Ce projet consiste à créer des téléphones portables au choix du client. En effet, celui-ci pourra choisir les différents composants de son téléphone et leur taille qui correspond à leur puissance.

3.2 Les classes

Chaque classe peut être vue comme un ensemble d'instances, d'individus. Une classe et ses sous-classes peuvent aussi être vues comme un concept. Chaque classe a une description qui précise les nécessités obligatoires pour qu'une instance puisse être membre de la classe. Une classe repose sur sa description. Il existe plusieurs types de descriptions comme le nommage, l'énumération des instances qu'elle contient ou encore les restrictions de propriétés que j'introduirai plus tard.

La classe Batterie par exemple contient uniquement des instances de batteries et non des coques ou des appareils photo, ce qui est assez logique, mais il faut obligatoirement définir tout ça. Une classe peut aussi être vue comme un domaine d'intérêt. La classe Batterie s'intéresse uniquement aux batteries. Il est possible de créer une hiérarchie de classes avec des classes mères et filles (ou superclasses et sous-classes).

Dans mon exemple, la classe appelée Composants aurait comme sous-classe Batterie, Appareilphoto, Mémoire, etc... Cela serait une hiérarchie très basique, que je pourrais compliquer mais le but ici est juste de présenter les bases. Je pourrais avoir une classe contenant des composants pour l'audio et la vidéo, une autre pour les composants hardware,

une autre pour les composants esthétiques, ... Ce serait une autre hiérarchie possible. La hiérarchie des classes est définissable comme on le souhaite mais doit être mûrement réfléchi.

Une propriété importante des classes est qu'elles peuvent être disjointes. Si deux classes sont disjointes, il est garanti qu'une instance d'une classe ne pourra pas être membre d'une classe disjointe! Par exemple, une instance de Batterie ne pourra pas faire partie de la classe Appareilphoto ou une instance de Memoire ne pourra pas non plus faire partie de la classe Bluetooth. Ces classes sont toutes disjointes entre elles.

Le terme "reasoner" a été mentionné précédemment. Il pourra être utilisé durant la création des classes. En effet, les relations entre les classes et leurs sous-classes seront automatiquement déduites par le reasoner. Son utilisation sera introduite et expliquée de manière plus concrète dans différents exemples plus tard lors de la création d'une ontologie. Voici un bout de code OWL qui pourrait définir ma classe Batterie où sont présents toutes les caractéristiques énoncées ci-dessus: déclaration d'une classe qui est une sous-classe et qui est disjointe avec d'autres classes.

```
1    <owl:Class rdf:ID="Batterie">      /**
2        <rdfs:subClassOf rdf:resource="#Composants"/>
3        <owl:disjointWith rdf:resource="#Appareilphoto"/>
4        <owl:disjointWith rdf:resource="#Memoire"/>
5        <owl:disjointWith rdf:resource="#Bluetooth"/>
6    </owl:Class>
```

Le but n'est pas d'écrire toute l'ontologie en OWL, car j'utilise Protégé pour la créer comme expliqué avant, mais il est très utile de comprendre le code correspondant afin d'effectuer des requêtes plus tard. Le gros défaut d'OWL et de RDF est que la lisibilité est vraiment compliquée. J'utilise des caractéristiques de RDF, de RDFS (RDF Schema) qui propose de hiérarchiser et décrire des classes et certaines d'OWL, ce qui est tout à fait possible étant donné qu'OWL est une extension de RDF et de RDFS. La classe est quelque chose de propre à OWL (owl:Class) mais on utilise une caractéristique de RDF par exemple pour définir son identifiant (rdf:ID).

Ce bout de code décrit la classe Batterie qui est une sous-classe de Composants

(`rdfs :subClassOf`) et elle est disjointe à la classe `Appareilphoto`, `Memoire` et `Bluetooth`. En effet, une batterie n'est pas un appareil photo, ni une mémoire, ni un bluetooth et ne peut donc pas être dans une de ses classes disjointes.

3.3 Les instances

Les instances sont des individus et sont membres d'une ou plusieurs classes. Elles respectent toutes les restrictions et la hiérarchie des classes. Si une instance fait partie de la classe `Batterie`, elle fera aussi partie de la classe `Composants` car `Batterie` est une sous-classe de `Composants`. Je vais donc pouvoir créer plusieurs instances qui vont être soit des objets, soit des actuateurs, soit des senseurs, des lieux, etc..., mais il est préférable de le faire une fois tous les concepts bien définis ainsi que leurs relations. Il est important de savoir qu'une instance est unique. Par exemple, le téléphone de Jean est une instance, celui de Luc en est une autre mais elles sont toutes deux des téléphones et peuvent appartenir à la même classe. Une instance peut être composée à son tour d'autres instances. Le téléphone de Luc par exemple est composé entre autres d'une batterie, d'un écran et d'une mémoire qui sont des instances.

3.4 Les propriétés

Il y a deux genres de propriétés: les propriétés de type de donnée (ou `data type properties` en anglais) et les propriétés d'objet (ou `object properties`). Les propriétés d'objet sont des relations binaires entre deux instances. La propriété *aBatterie* relie par exemple une instance `Smartphone` et une instance `Batterie`. La propriété *appartientA* serait certainement la relation entre une instance `Smartphone` et une instance `Personne`. Les propriétés d'objet pourront être fonctionnelles, transitives, symétriques ou encore d'un autre type. Une propriété peut relier deux instances d'une même classe ou de classes différentes.

Quant à elles, les propriétés de type de donnée (ou `data type properties`) sont des relations binaires entre une instance et un type de donnée (`int`, `float`, `date`, ...). La propriété *estCouleur* définit la couleur d'un composant comme une coque de type `string`. Une coque par exemple aura cette propriété pour définir sa couleur qui pourra être bleu, vert, jaune, rouge, etc...

La propriété fonctionnelle et fonctionnelle inverse:

Une propriété est dite fonctionnelle si pour une instance donnée il y a au plus une instance qui est en lien avec celle-ci. Elle est aussi appelée propriété à valeur unique. La propriété *aBatterie* est fonctionnelle car chaque instance de Smartphone n'est liée qu'à une instance de Batterie. Si une instance de Smartphone est liée à deux instances de Batterie, les deux instances de Batterie doivent être identiques. Si elles ne le sont pas, l'ontologie sera inconsistante. C'est-à-dire qu'elle ne respecte pas les règles. L'inconsistance d'une ontologie sera étudiée plus tard.

C'est à ce moment qu'il faut bien différencier les classes et les instances. Une instance est unique comme je l'ai déjà dit. Il y a plusieurs mêmes types de batterie, mais chacune aura un identifiant ou/et un nom différent par exemple. Les propriétés s'appliquent uniquement aux instances et indirectement aux classes.

Une propriété est fonctionnelle inverse si sa propriété inverse est une propriété fonctionnelle. Si on reprend notre exemple juste au-dessus, la propriété *estBatterie* est une propriété fonctionnelle inverse. En effet, c'est la propriété inverse de *aBatterie*. Cela fonctionne exactement comme avant. Si une instance de Batterie est liée à deux instances de Smartphone, ces deux instances doivent être identiques sinon l'ontologie sera inconsistante.

La propriété transitive

Si une propriété "a" est liée à une propriété "b" et "b" est liée à une propriété "c", on dit que la propriété est transitive. Dans mon exemple d'ontologie, la propriété *aComposant*, qui contient les propriétés *aBatterie*, *aMemoire* et d'autres est transitive. En effet il y a un lien transitif entre le smartphone, le composant et la mémoire ou la batterie par exemple. Smartphone a un composant et ce composant possède une batterie ou une mémoire.

Dans le tutoriel de Manchester, la propriété transitive est expliquée avec les liens généalogiques. La propriété *hasAncestor* est transitive car si Matthew a comme ancêtre Peter et que Peter a comme ancêtre William, transitivement, Matthew a comme ancêtre William.

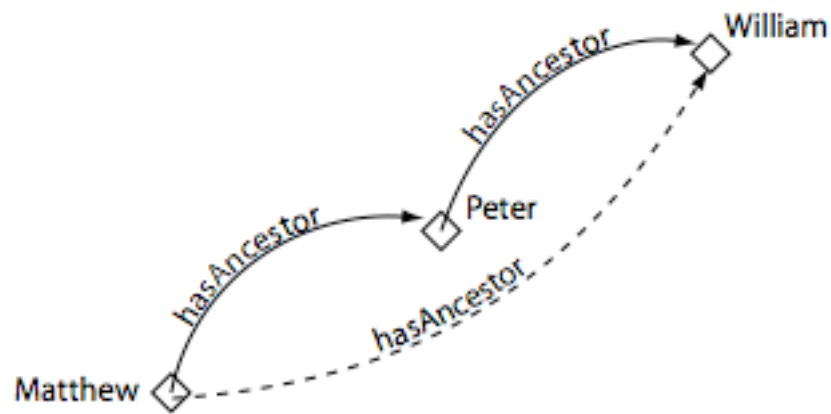


Figure 2: Illustration d'une propriété transitive (pris dans [6])

La propriété symétrique et asymétrique

Si une propriété “a” est liée à la propriété “b” et que “b” est aussi liée à “a”, cette propriété est symétrique comme son nom l’indique. L’ontologie Smartphone n’a pas de fonction symétrique. Une propriété *aSoeurFrère* pourrait être créée pour les instances de type *Personne*, mais il n’y a aucun intérêt dans cet exemple de modélisation.

Afin d’illustrer cette propriété, l’exemple et la figure 6 sont tirés du tutoriel de Manchester [6] qui présente la propriété *hasSibling* comme symétrique. En effet si Matthew a comme frère et soeur Gemma, Gemma a aussi Matthew comme frère et soeur. C’est une propriété qui fonctionne dans les deux sens.

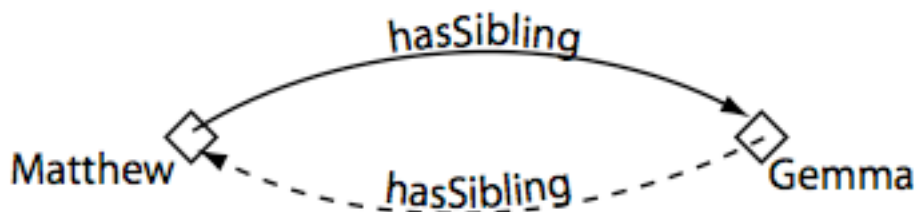


Figure 3 : Illustration d'une propriété symétrique (pris dans [6])

Une propriété est asymétrique si elle n'est pas symétrique. Etant donné que dans mon ontologie, il n'y a aucune propriété qui est symétrique, elles peuvent toutes être asymétriques.

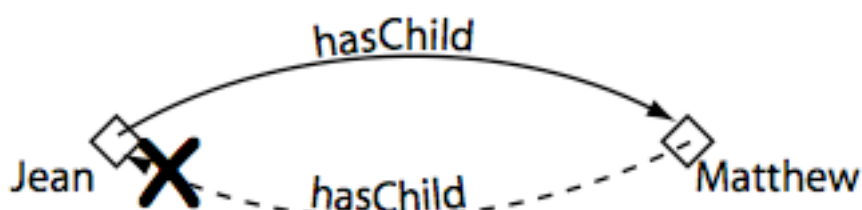


Figure 4 : Illustration d'une propriété asymétrique (pris dans [6])

La propriété réflexive et irreflexive

Une propriété est dite réflexive quand est en lien avec elle-même.

L'exemple a aussi été repris du tutoriel de l'Université de Manchester. La propriété *knows* est réflexive. En effet, une personne se connaît elle-même en plus de connaître d'autres personnes.

Comme on le voit sur l'image ci-dessous, George se connaît lui-même mais connaît aussi Simon.

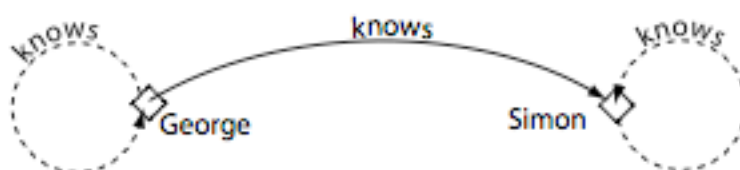


Figure 5 : Représentation imagée d'une propriété réflexive (Source [21])

Comme dans toutes les autres types de propriétés, une propriété est dite irreflexive si elle est l'inverse d'une propriété réflexive. Comme pour les propriétés asymétriques, elles sont toutes irreflexives dans notre ontologie.

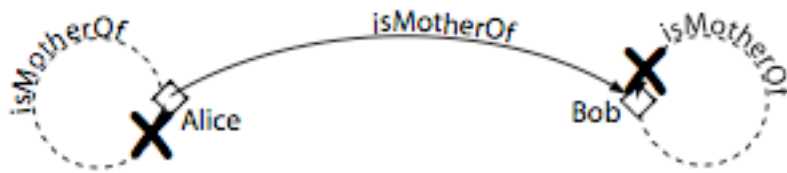


Figure 6 : Illustration d'une propriété irréflexive (pris dans [6])

3.5 Création d'une ontologie

La première chose à faire est de créer l'ontologie "Smartphone". Tout va se passer dans l'outil Protégé. Une fois Protégé ouvert, il suffit de créer un nouveau projet en lui donnant si possible un nom adéquat. J'ai fait très simple et l'ai nommée Smartphone.

Il est important de faire les choses dans un ordre logique. La première étape est de créer les classes avant les propriétés et les instances. Il faut donc gérer toutes les classes et leur(s) sous-classe(s) une par une. Ensuite, ce sera la création des propriétés et finalement des instances.

3.5.1 Création des classes

Maintenant que les principaux éléments d'une ontologie, ont été évoqués et expliqués, il faut évidemment commencer par créer les concepts, donc les différentes classes. Comme mentionné précédemment, Protégé a plusieurs onglets. La création des classes se fait uniquement dans l'onglet OWLClasses. Les classes sont toujours représentées en jaune dans Protégé.

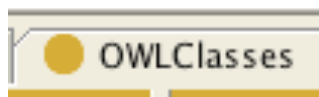


Figure 7 : L'onglet Classes dans Protégé

Il y a trois boutons pour gérer les classes: le premier pour créer une nouvelle classe, le second pour créer une sous-classe et le troisième pour supprimer la ou les classes sélectionnées.



Figure 8 : Les trois boutons pour gérer les classes dans Protégé

Il est bien de choisir une façon de nommer ses classes et de la respecter. En effet, si après on veut utiliser notre ontologie et faire des requêtes, il est préférable que toutes les classes soient écrites de la même façon ainsi que les propriétés et les instances évidemment. Généralement, on met en majuscule la première lettre d'une classe. Il faut bien choisir les noms des classes afin qu'elles soient éloquentes, que l'on comprenne immédiatement les individus qu'elles pourraient posséder, ce qu'elles représentent.

Les classes vont donc former une hiérarchie et se décomposer sous la forme d'un arbre. La racine est le nom donné à une ontologie; dans mon exemple: Smartphone. Chaque concept est une classe différente et est composée d'une ou plusieurs sous-classes, qui elles-mêmes pourront posséder des sous-classes et ainsi de suite. Voir une hiérarchie de classe comme un arbre peut aider à la compréhension d'une ontologie.

La classe Thing est la classe qui représente l'ensemble des instances. Donc toutes les classes sont théoriquement des sous-classes de Thing en commençant par la classe Smartphone.

Comme mentionné précédemment, un smartphone sera composé de plusieurs éléments, qui correspondent à différentes classes qu'il faut créer (appareil photo, mémoire, batterie, Bluetooth, etc...).

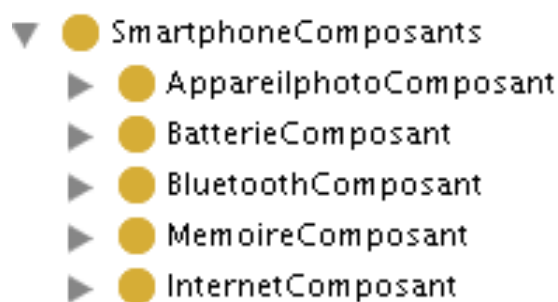


Figure 9 : Hiérarchie de la classe SmartphoneComposants sous Protégé

Chaque sous-classe de Composants possède elle aussi des sous-classes. En effet, BatterieComposant possédera plusieurs types de batterie. Il en sera de même pour tous les

autres composants dans cet exemple, mais ce n'est pas obligatoire. Je pourrais avoir uniquement une classe BluetoothComposant et proposer les divers types de bluetooth à choix grâce à des propriétés de donnée. Cela serait une autre manière de faire.

Chaque sous-classe de Composants est disjointe avec toutes les autres comme expliqué précédemment. Cette action s'effectue toujours dans l'onglet OWLClasses de Protégé dans la partie de droite tout en bas. Il y a plusieurs options (boutons):

- le premier crée une nouvelle classe disjointe grâce à une expression OWL.
- le second ajoute une classe existante comme disjointe.
- le troisième ajoute toutes les classes sibling (même niveau) comme étant disjointes.
- le quatrième supprime toutes les classes sibling et elles ne seront plus disjointes.

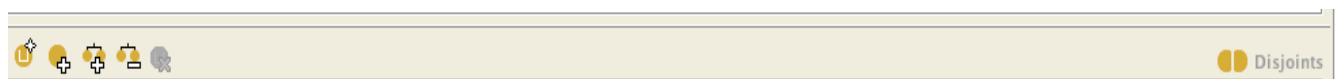


Figure 10 : Les différentes actions possibles pour une classe disjointe dans Protégé

Protégé est un outil qui a le sens de l'anticipation. Si on sélectionne la classe BatterieComposant et qu'on met toutes les classes "siblings" comme disjointes, Protégé va donc savoir que chaque classe "sibling" a aussi ses classes "sibling" comme disjointes. C'est-à-dire que Protégé va par exemple définir que les classes disjointes de MemoireComposant sont AppareilphotoComposant, BatterieComposant, InternetComposant et BluetoothComposant.

Cela évite qu'on ait besoin de le faire pour chaque classe, ce qui serait très embêtant pour de très grandes ontologies.

La figure 11 montre les classes disjointes de BatterieComposant dans l'interface Protégé:

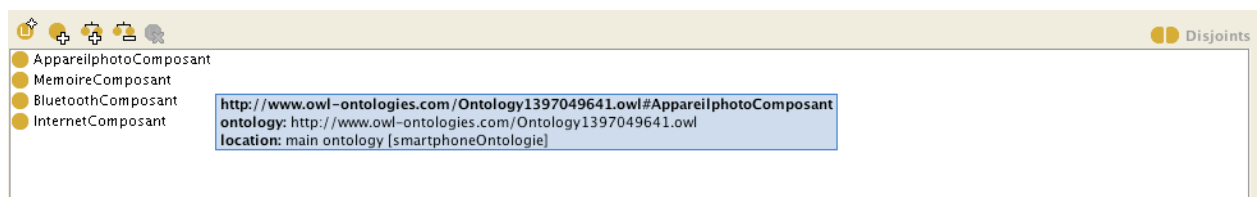


Figure 11 : Affichage des classes disjointes dans Protégé

Maintenant, il faut créer les sous-classes de chaque composant. Comme le but est que le client puisse choisir ses composants, il faut qu'il puisse également choisir le type. Par exemple, certains voudront une batterie très performante, d'autres préféreront miser sur un appareil photo de très grande qualité, d'autres voudront les deux. L'objectif est que le client puisse choisir ses composants comme il le veut.

On peut par exemple créer différents appareils photo selon leur qualité, soit les pixels. Cinq classes ont été créées, une pour chaque qualité soit SixMPAppareilphoto, HuitMPAppareilphoto, DixMPAppareilphoto, DouzeMPAppareilphoto et SeizeMPAppareilphoto.

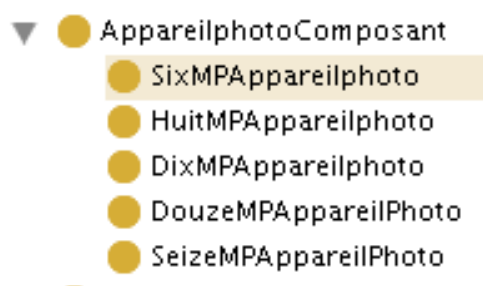


Figure 12 : Les sous-classes d'AppareilphotoComposant

Trois classes représentent les différents types de mémoires possibles où la capacité est la notion la plus importante.



Figure 13 : Les sous-classes de MemoireComposant

Toutes ces sous-classes sont disjointes entre elles. En effet, une instance de PetiteMemoire ne peut pas appartenir à la classe MoyenneMemoire tout comme SixMPAppareilPhoto ne peut pas appartenir à la classe SeizeMPAppareilphoto.

La figure 14 montre la hiérarchie des classes une fois que toutes les sous-classes de chaque composant ont été créées.

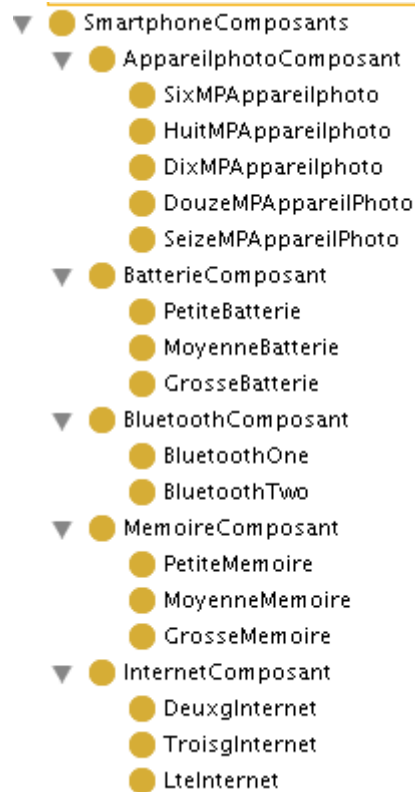


Figure 14 : Hiérarchie complète de la classe SmartphoneComposants

Pour complexifier un peu cette ontologie, un autre concept a été créé: la protection du téléphone. En effet, quand on achète un téléphone, on aime bien acheter une coque ou un étui afin de le protéger des chutes, des rayures ou simplement pour le rendre plus beau.

La classe SmartphoneProtection a comme sous-classe CoqueProtection et EtuiProtection.

Comme pour les composants, des sous-classes pour les différents types de coque et d'étui ont été créés.

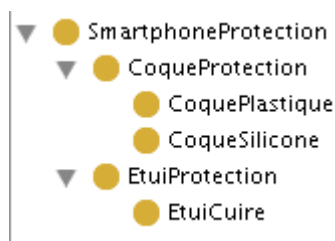


Figure 15 : Hiérarchie de la classe SmartphoneProtection

Cette ontologie pourrait être beaucoup plus complexe en y ajoutant de nouveaux concepts. Le but ici est juste de présenter les principales caractéristiques d'OWL et de Protégé pour créer une ontologie simple et basique tout en ayant une architecture réfléchiée par composant et types de composant. Toutes ces classes sont aussi disjointes entre elles. CoqueProtection est

disjointe de EtuiProtection et CoquePlastique est disjointe de CoqueSilicone.

3.5.2 Création des propriétés

Comme mentionné précédemment, il y a deux types de propriétés: les propriétés d'objets et les propriétés de données. Il faut donc commencer par créer les propriétés d'objets qui relient deux instances. A noter qu'OWL possède un troisième type de propriété : les propriétés d'annotation ou Annotation Properties. Elles jouent le rôle de métadonnées. Elles servent à ajouter de tierces informations sur des classes, des instances ou sur des propriétés elles-mêmes comme un commentaire, le nom de l'auteur, la date, etc... Je n'ai pas jugé nécessaire de le faire pour cet exemple d'ontologie, mais il est préférable de le faire pour une ontologie plus au moins compliquée.

La création des propriétés d'objets se fait dans l'onglet Object Properties. Elles sont représentées en bleu dans Protégé.

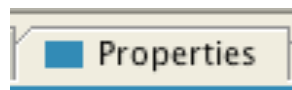


Figure 16 : Onglet Properties dans Protégé

Tout comme les classes, il y a trois boutons pour gérer les propriétés d'objets: le premier pour créer une nouvelle propriété, le second pour créer une sous-propriété à la propriété sélectionnée et le troisième pour supprimer le ou les propriétés sélectionnées.

Il est aussi bien de respecter une norme pour nommer ses propriétés. Elles sont souvent composées en premier d'un verbe, puis d'un nom. Personnellement je commence le verbe avec une minuscule et le nom avec une majuscule. Mieux vaut encore une fois bien choisir les noms, car ça sera plus facile de s'y retrouver que si on appelle ses propriétés: *propriété1*, *propriété2*, ...

Dans l'ontologie Smartphone, chaque smartphone peut posséder plusieurs composants. Il y a un lien d'appartenance entre une instance de Smartphone et une instance de Composants. Ces relations sont des propriétés d'objet. Il faut donc créer une propriété d'objet pour chaque type de composants.

Chaque propriété peut avoir une propriété inverse si la relation est à double sens. Dans ce cas-ci, chaque instance d'un composant appartient à une instance de Smartphone. Toutes les

propriétés qui lient les composants au smartphone ont une propriété inverse. Il ne faut pas oublier de les créer. Pour cela, il suffit de créer la propriété inverse ou d'ajouter une déjà existante tout en bas à droite toujours dans l'onglet Object Properties.



Figure 17 : Créer ou définir une propriété d'objet inverse dans Protégé

Il est également possible de hiérarchiser les propriétés comme les classes. Dans cet exemple, j'ai créé une propriété *aComposant* qui est la relation entre le smartphone et un composant. Cette propriété contient comme sous-propriétés les relations entre le smartphone et chaque type de composant (batterie, mémoire, internet,...)

Une autre propriété est *aProtection* qui est la relation entre un smartphone et sa protection. Ses sous-propriétés sont *aCoque* et *aEtui*.

Voici un extrait d'une déclaration d'une propriété d'objet en OWL:

```
1    <owl:ObjectProperty rdf:ID="aBatterie">   /**
2        <rdfs:domain rdf:resource="#Smartphone"/>
3        <rdfs:range rdf:resource="#BatterieComposant"/>
4    </owl:ObjectProperty/>
```

En OWL, la classe `owl:ObjectProperty` regroupe les propriétés dont la valeur est une ressource.

Créer les propriétés inverses n'est pas obligatoire mais c'est utile dans certains cas pour créer des contraintes supplémentaires grâce aux fonctions. Elles seront d'ailleurs créées dans l'ontologie de la Smart Home.

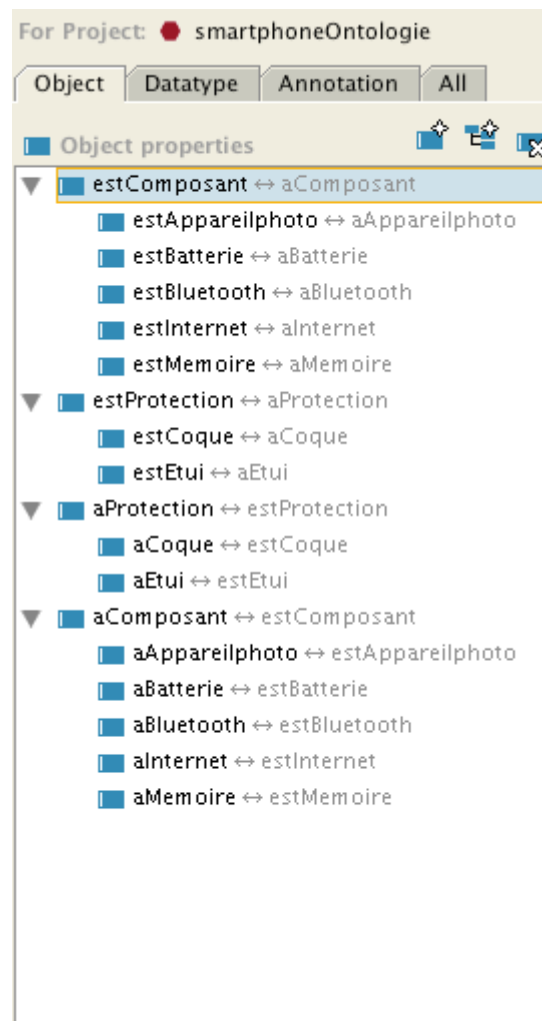


Figure 18 : Hiérarchie des propriétés d'objets pour l'ontologie Smartphone

La propriété inverse d'une propriété est affichée en gris à côté de celle-ci comme le montre l'image ci-dessus.

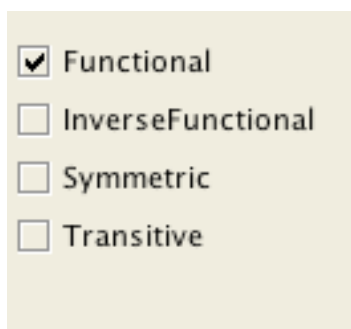
On peut enrichir le sens d'une propriété en lui assignant une caractéristique grâce à OWL.

Dans Protégé on le fait via les checkboxes qui se situent dans le panel de droite au dessus de la fonction inverse. Elles permettent de définir la ou les caractéristiques d'une propriété d'objets. Dans la version 3.4.7 de Protégé, une propriété peut avoir comme caractéristique d'être fonctionnelle, inversement fonctionnelle, transitive ou symétrique. Il est très important de bien comprendre la différence entre toutes ces caractéristiques que j'ai expliquées dans le chapitre 3.4. Dans la version 4 de Protégé, de nouvelles caractéristiques sont apparues comme l'asymétrique, la réflexive et la non réflexive (irreflexive en anglais).

Durant ce travail, ces nouvelles caractéristiques ne sont pas nécessaires t donc ça ne pose aucun problème de travailler avec une ancienne version de Protégé.

La propriété *aBatterie* est fonctionnelle comme expliqué dans la présentation de cette fonction. Il suffit de sélectionner la propriété *aBatterie* et de cocher la case “Functional”. Protégé anticipe bien et coche automatiquement la case “InverseFunctional” pour la propriété *estBatterie*. En effet, si *aBatterie* est fonctionnelle, sa propriété inverse est logiquement inversement fonctionnelle.

Toutes les sous-propriétés de *aComposant* sont fonctionnelles et leur propriété inverse est fonctionnelle inverse. Les propriétés *aComposant* et *aProtection* sont quant à elles transitives.



<input checked="" type="checkbox"/>	Functional
<input type="checkbox"/>	InverseFunctional
<input type="checkbox"/>	Symmetric
<input type="checkbox"/>	Transitive

Figure 19 : Définir une propriété d’objet comme étant fonctionnelle

Entre le panel qui affiche la hiérarchie des propriétés et celui qui affiche les caractéristiques, on peut spécifier le “domaine” et le “range” de la propriété sélectionnée. Les propriétés lient les entités du domaine aux entités du range. En OWL, c’est important de bien comprendre que les domaines et les ranges ne sont pas des contraintes, mais des classes. On peut alors définir à quelle(s) classe(s) doivent obligatoirement appartenir les instances d’une propriété. On les utilise comme axiomes pour le reasoner qui sera donc capable d’utiliser ces informations pour nous indiquer si une ontologie est consistante ou non. Le point négatif est que le reasoner pourra parfois proposer des erreurs de classification étranges. De plus, il n’est pas précis du tout dans ses indications d’erreurs. Cela dépend quel reasoner on utilise, mais la plupart ne donnent que peu d’informations sur l’erreur et il est donc difficile de déboguer une ontologie inconsistante. C’est pour cela que pour de grandes ontologies, où il sera difficile de localiser l’erreur, il n’est pas forcément conseillé de remplir le domaine et le range.

Dans cet exemple, la propriété *aBatterie* lie une instance de smartphone et une instance de batterie. Le domaine de *aBatterie* est donc Smartphone et le range la BatterieComposant.

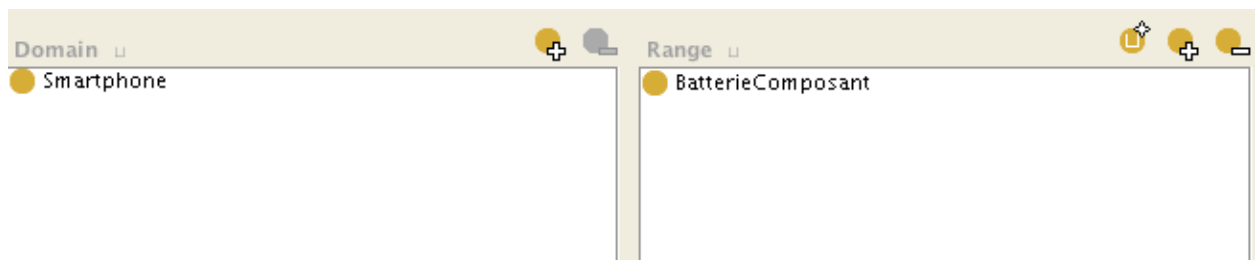


Figure 20 : Le domaine et le range de la propriété *aBatterie*

A noter qu'il est possible de spécifier plusieurs classes comme range. Protégé va interpréter ça comme une union de classes. Etant donné que toutes les classes sont disjointes entre elles dans cette ontologie, il n'est pas possible de le faire.

Protégé est capable de trouver automatiquement le domaine et le range de la propriété inverse. Effectivement, la propriété *estBatterie* aura comme domaine BatterieComposant et comme range Smartphone, car une instance de batterie appartiendra à une instance de smartphone. Normalement le domaine d'une propriété correspond au range de son inverse et le range au domaine de son inverse.

3.5.3 Création des restrictions

Une fois que toutes les propriétés d'objet ont été créées, il faut maintenant les utiliser pour décrire les classes créées précédemment au moyen de restrictions. En effet pour l'instant, l'ontologie est très globale et permet énormément de choses. Il faut donc restreindre les choix. Par exemple, on peut choisir qu'un smartphone peut posséder plusieurs batteries pour le moment, ce qui n'est pas idéal. Le but est donc de restreindre certaines caractéristiques des classes. Une restriction décrit donc une classe et se base sur la ou les relations entre elles qui ont été définies précédemment. Il faut la voir comme une condition. Elle décrit une classe qu'on appelle anonyme. Cette classe contient toutes les instances qui satisfont cette restriction. L'idée est qu'une classe d'instances est définie par les relations auxquelles ses instances participent.

Voici quelques exemples de restrictions qui pourraient être créées:

- Une classe contenant tous les smartphones qui ont au moins une coque.
- Une classe contenant tous les smartphones qui ont exactement une batterie et une mémoire. Cela serait le téléphone le plus basique.
- Une classe contenant tous les smartphones qui ont la batterie la plus performante et la

mémoire la plus performante

- Etc ...

Le but est de créer quelques restrictions afin de présenter quelques possibilités que nous offre les restrictions en OWL.

OWL propose plusieurs types de restrictions pour les propriétés d'objets. On peut les classer en deux grandes catégories :

1. les restrictions de quantité
2. les restrictions de cardinalité

Pour créer une restriction, il faut se rendre dans l'onglet des classes, sélectionner la classe voulue et utiliser le panel du milieu où il y a inscrit "Asserted Conditions". On retrouve plusieurs boutons. Les deux premiers permettent de créer une restriction. Le quatrième permet de supprimer la restriction sélectionnée.

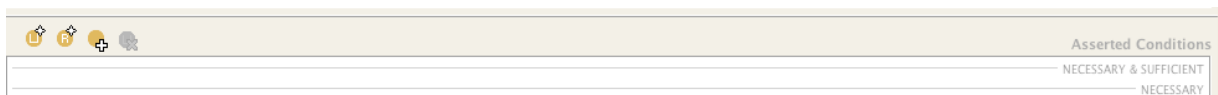


Figure 21 : Panel pour créer une restriction

Restrictions de quantité

Les restrictions de quantité sont classées en deux catégories: les restrictions existentielles et universelles. « Une restriction existentielle décrit la classe des individus pour lesquels au moins une valeur d'une certaine propriété est instance de la classe ». [5] Dans cet exemple, un smartphone a au moins une relation *aProtection* si on part du principe qu'un client peut avoir plusieurs protections de même type ou non. Dans Protégé, on utilise le mot clé "some" pour représenter cette restriction. En OWL, les restrictions existentielles sont connues sous le nom de "SomeValuesFrom restrictions" et peuvent être représentées par le signe « il existe » : \exists .

Les restrictions existentielles spécifient l'existence d'au moins une relation d'une propriété à une instance qui est membre de la classe spécifiée. Cependant, cela ne veut pas dire que ce

sont les seules relations pour cette propriété qui peuvent exister pour les instances de cette classe.

La sous-classe de Smartphone nommée ProtegeSmartphone va contenir uniquement des instances de smartphone qui ont au moins une protection. Une restriction existentielle doit être créée pour définir cela.

Il faut donc préciser la propriété (aProtection) suivi de la restriction existentielle (some) et de la classe concernée (SmartphoneProtection). Il y a deux moyens de créer cette restriction dans Protégé, soit avec le premier bouton (create new expression) ou avec le second (create restriction).

Avec le premier bouton, on sélectionne la propriété que l'on souhaite, ici *aProtection*, puis le type de la restriction à l'aide des boutons, ici le second depuis la gauche, ou en écrivant directement dans l'input le mot clé « some », puis en écrivant la classe SmartphoneProtection. Le rond jaune avec un \forall représente la restriction existentielle some. La figure 22 montre la création d'une restriction avec Protégé.

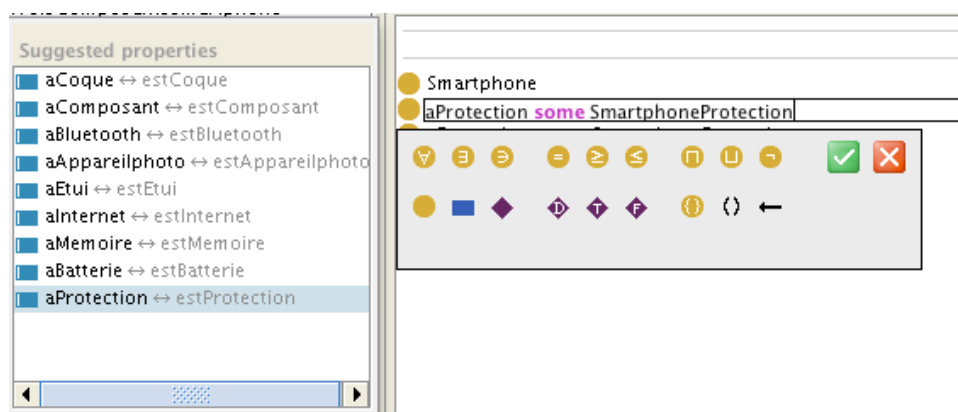


Figure 22 : Création d'une nouvelle expression dans Protégé

Avec le second bouton, une fenêtre s'ouvre. On sélectionne la propriété voulue à gauche, le type de restriction à droite et en bas la classe ou les classes. D'autres restrictions permettent de définir une valeur par exemple à la place d'une classe.

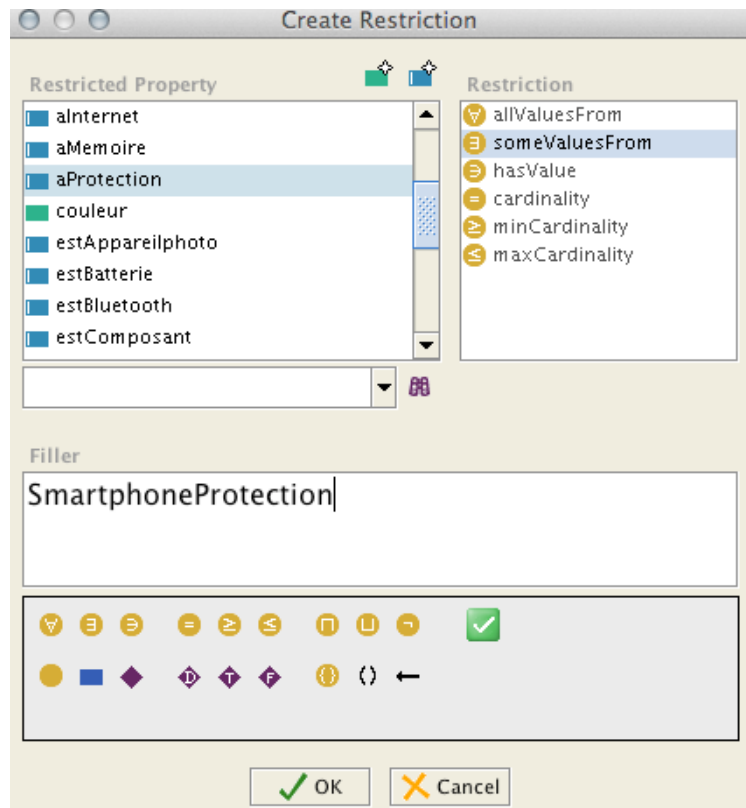


Figure 23 : Création d'une restriction

● Smartphone
 ⊃ aProtection **some** SmartphoneProtection

Figure 24 : Affichage d'une restriction existentielle

« Une restriction universelle décrit la classe des instances pour lesquels toutes les valeurs d'une certaine propriété sont des instances de la classe. » [5] On utilise le mot clé “only” dans Protégé. En OWL, les restrictions universelles sont connues sous le nom de “AllValuesFrom restrictions” en OWL et peuvent être représentées par le signe \forall .

La classe *MemoireSmartphone* par exemple utilise cette condition “only”. Elle comporte des instances qui ont la mémoire la plus grande, donc qui font partie de la classe *GrosseMemoire* en utilisant la propriété *aMemoire*. Cela signifie que toutes les instances de cette classe satisfont cette restriction, mais cela ne veut pas dire que les instances auront uniquement une grosse batterie. On restreint uniquement la propriété citée. Si on ne met que cette restriction, les instances de cette classe pourront utiliser les autres propriétés sans aucune restriction. Ces

instances pourront aussi avoir en plus d'une grosse mémoire, un appareil photo, une batterie, etc...



Figure 25 : Affichage d'une restriction universelle

Le smartphone classique quant à lui peut posséder uniquement un appareil photo, une batterie ou une mémoire. Cela ne veut pas dire qu'il peut avoir seulement un de ces trois composants, mais il ne peut pas avoir d'autres composants que ceux-ci. Il est très important de comprendre que c'est une union de composants. Une faute serait de mettre une intersection (and) à la place d'une union (or). Le reasoner indiquerait une erreur si ces classes sont disjointes, ce qui est le cas, car une instance ne peut pas faire partie de la classe BatterieComposant et MemoireComposant.

Il faut utiliser la propriété *aComposant* avec la restriction *only* et les classes BatterieComposant, MemoireComposant et AppareilphotoComposant.



Figure 26: Affichage d'une seconde restriction universelle

Toutes les restrictions utilisées jusqu'à présent sont des conditions nécessaires. C'est à dire que si une instance appartient à une classe qui a une ou des restrictions, il est nécessaire que cette instance remplisse ces conditions, mais cela ne marche que dans ce sens. On ne peut pas dire que si une instance remplit ces conditions, elle fait automatique parti de cette classe. Cela ne suffit pas. Il faut que les conditions soient suffisantes en plus d'être nécessaires. Une classe qui a uniquement des conditions nécessaires est appelée une classe primitive. Une classe qui a des conditions nécessaires et suffisantes est appelée une classe définie.

Dans Protégé 4, on peut convertir une classe primitive en une classe définie, ce qui n'est pas possible avec la version 3 de Protégé, mais cela ne pose pas de problème étant donné que il n'y a pas besoin d'avoir des classes définies pour ce projet.

Voici le code OWL de la classe ClassiqueSmartphone:

```
<owl:Class rdf:ID="ClassiqueSmartphone">
[...]
  <owl:Restriction>
    <owl:onProperty rdf:resource="aComposant"/>
    <owl:allValuesFrom>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class
rdf:about="#AppareilphotoComposant"/>
        <owl:Class
rdf:about="#BatterieComposant"/>
        <owl:Class
rdf:about="#MemoireComposant"/>
      </owl:unionOf>
    </owl:Class>
  </owl:allValuesFrom>
</owl:Restriction>
[...]
```

Une restriction est représentée par une ressource de type owl:Restriction qui est sujet d'une propriété owl:onProperty qui possède la valeur de la propriété de la restriction (allValuesFrom ou someValuesFrom).

Restrictions de cardinalité

En OWL et donc dans Protégé, on peut définir des classes d'instances qui ont au moins, au plus ou exactement un nombre spécifique de relations que ça soit des propriétés d'objets ou de données. Une restriction de cardinalité minimum indique le nombre minimum de relations qu'une instance peut avoir. Au contraire une restriction de cardinalité maximum indique le nombre maximum de relations qu'une instance peut avoir. Une restriction de cardinalité exacte définit le nombre exact de relations qu'une instance peut posséder. On utilise les mots-clés min, max et exactly dans Protégé et les propriétés owl:minCardinality, owl:maxCardinality et owl:cardinality en OWL.

Dans cette ontologie, la classe TroisComposantsSmartphone contient des instances qui ont exactement trois composants. Pour créer cette restriction, il faut utiliser la propriété *aComposant*, le mot clé exactly et le chiffre 3.



Figure 27 : Affichage d'une restriction de cardinalité

3.5.4 Création des propriétés de données

Comme dit dans le chapitre 3.4, les propriétés de données ou datatype properties lient une instance à une valeur d'un type de donnée de XML Schema ou à une littérale RDF. Elles décrivent donc la relation entre une instance et des valeurs de données.

Dans Protégé, on utilise l'onglet Datatype dans Properties:

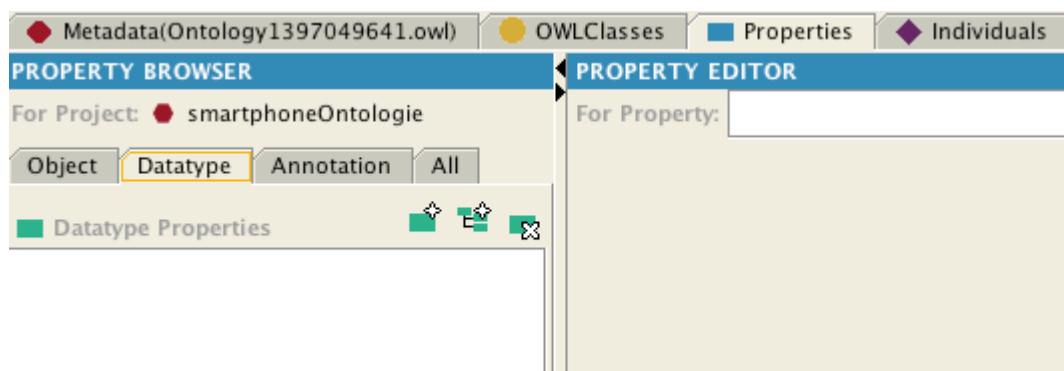


Figure 28 : Onglet pour les propriétés de données dans Protégé

Une propriété de donnée peut être utilisée afin de spécifier la marque des composants existants par exemple. Une instance de `AppareilphotoComposant` pourrait être de marque Nokia, Samsung, Apple, etc... Tout comme les classes et les propriétés d'objets, on peut classer les propriétés de données en hiérarchie en créant des sous-propriétés. C'est toute la force d'une ontologie.

Comme les propriétés d'objets, les propriétés de données ont aussi un domaine et un range. Le domaine est défini par une ou plusieurs classes, mais pas le range. En effet, elles relient une instance à une donnée. Le range est donc un type de donnée comme un string, un integer, un float, une date etc...

Chaque composant doit être disponible dans des marques spécifiques. Les marques ne sont pas identiques pour tous les composants. Il faut donc créer une propriété pour chaque composant afin de définir la marque du composant. La figure 29 affiche ces propriétés pour les marques.

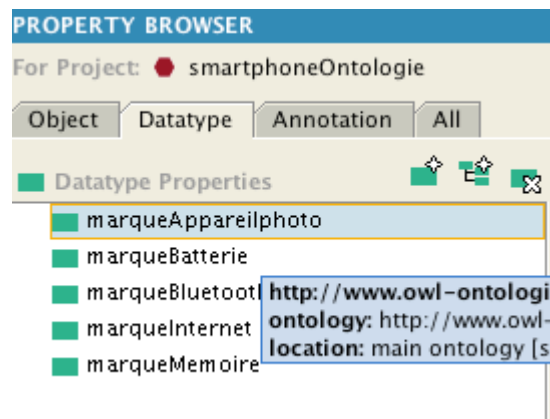


Figure 29 : Les propriétés de données pour définir les marques de chaque composant

Il faut ensuite aussi préciser le domaine et le type du range. Pour la propriété 'marqueBatterie', le domaine est évidemment 'BatterieComposant' et le range est un string. Etant donné que le client peut choisir, il faut lui donner les choix disponibles. Deux marques seulement sont proposées pour les batteries: Samsung et Apple.

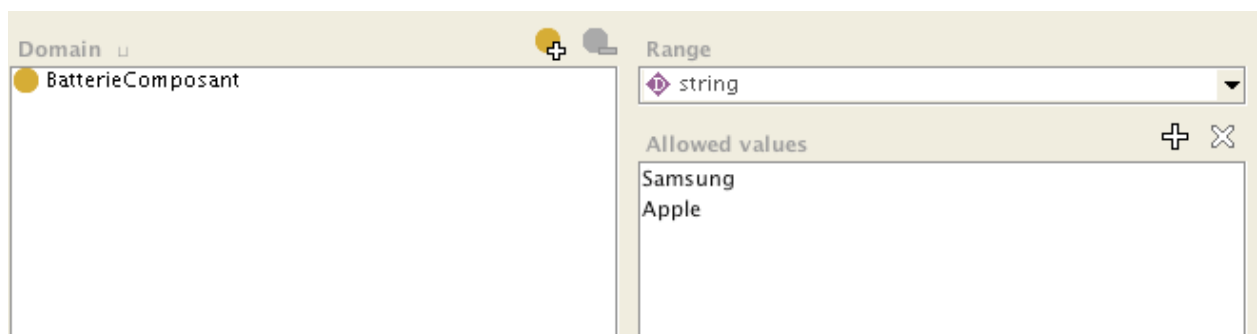


Figure 30 : Choix du domaine et du range pour une propriété de données

Un second type de donnée appelé "couleur" est utilisé pour que le client puisse choisir la couleur de son étui et de sa coque.

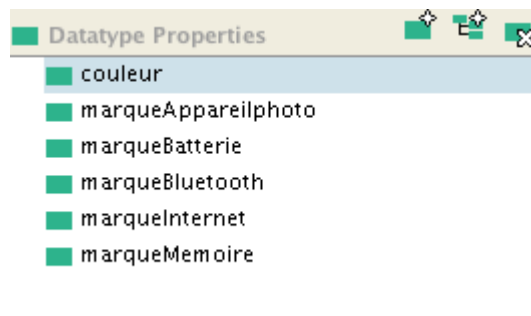


Figure 31 : Création de la propriété de données « couleur »

Tout type de protection est disponible dans les mêmes couleurs. Afin de bien utiliser la hiérarchie de classes, le domaine ne sera pas les classes *CoqueProtection* et *EtuiProtection*, mais la super-classe *SmartphoneProtection*. Le but d'une ontologie est de pouvoir ajouter des concepts, des classes, des propriétés facilement. Si je décide de proposer un nouveau type de protection, les instances de cette classe hériteront des propriétés de leur super-classe et donc de la propriété couleur et seront disponibles directement dans les mêmes couleurs que les coques et les étuis. Ces couleurs sont les suivantes : bleu clair, cyan, blanc, noir, magenta, jaune et orange comme le montre la figure 32.

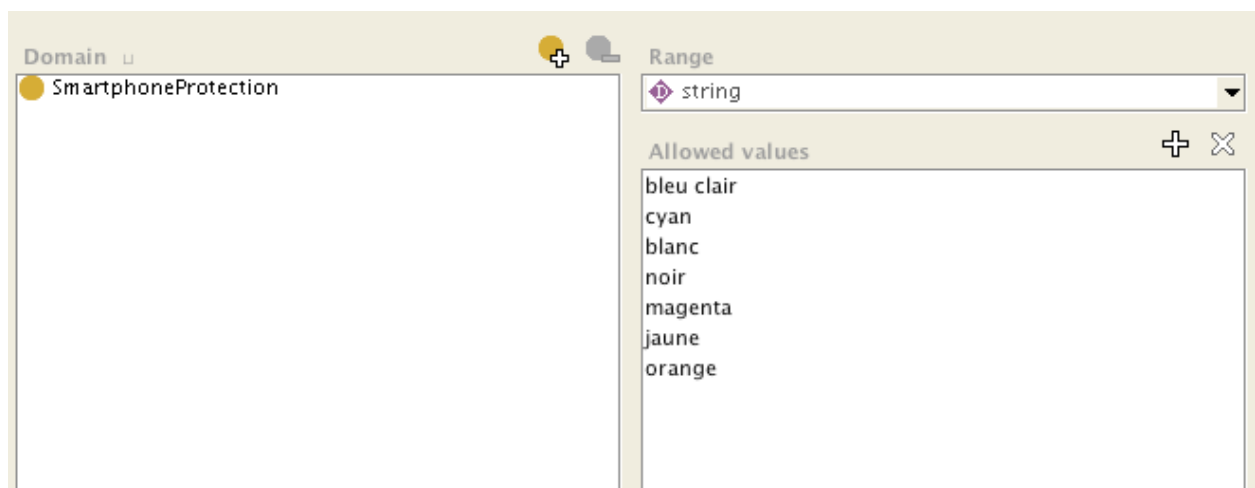


Figure 32 : Choix du domaine et du range pour la propriété couleur.

3.5.5 Création des instances

Place maintenant à la création des instances. Le client ne va pas avoir accès à Protégé et il faudra lui proposer une autre interface pour lui permettre de créer son propre smartphone avec les composants de son choix. Il est possible par contre de lui proposer des smartphones déjà tout prêts grâce aux instances. Cela se passe dans l'onglet "Individuals" de Protégé. Les instances sont représentées par des losanges violets.

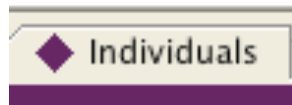


Figure 33 : Onglet pour créer des instances dans Protégé

Il y a trois panels qui s'affichent dans cet onglet. Celui de tout à gauche est pour sélectionner la classe des instances, celui du milieu affiche les instances de la classe sélectionnée et enfin celui de droite affiche l'instance sélectionnée.

Pour créer une instance, il faut d'abord sélectionner la classe de l'instance dans le panel de gauche. Il y a les mêmes boutons que pour les classes et les propriétés pour gérer la création et la suppression des instances. Le plus simple est de commencer par créer une instance de `TroisComposantsSmartphone`, puis de créer les instances qui le composeront plutôt que de créer des composants et de les affilier ensuite à une instance de `TroisComposantsSmartphone`. Une fois cette instance créée, elle s'affiche dans le panel de droite. L'instance va respecter toutes les restrictions définies auparavant et proposer uniquement ce qui est possible de lier à cette instance.

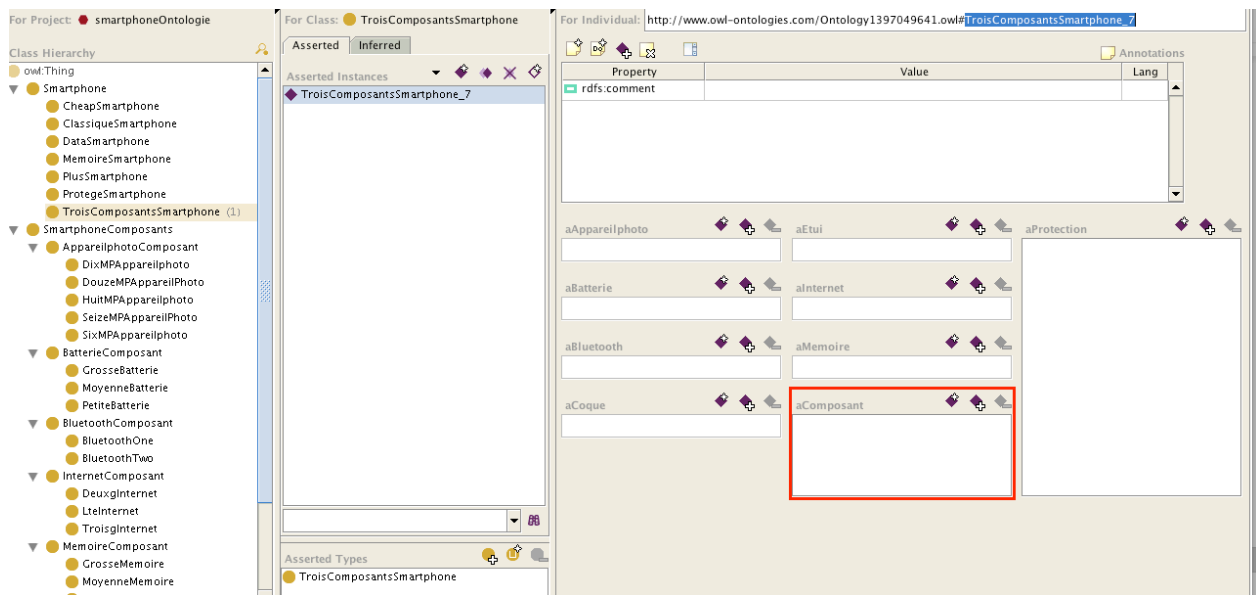


Figure 34 : Protégé affiche les restrictions non respectées dans un encadré rouge

Il faut maintenant créer d'autres instances qui seront liées via les différentes propriétés créées précédemment. Pour le *TroisComposantsSmartphone*, la seule contrainte existante est qu'il doit au moins avoir trois composants via la propriété *aComposant*. Protégé entoure en rouge les contraintes qui ne sont pas encore respectées comme le montre la figure 34. J'ai choisi un appareil photo de douze mégapixels pour mon smartphone. J'ai donc créé une instance de *DouzeMPAppareilphoto* en utilisant le bouton prévu à cet effet juste au dessus de *aComposant*. A chaque création d'une instance, une nouvelle fenêtre s'ouvre comme une pop-up. On aura accès à toutes les propriétés d'objets et de données de cette instance.

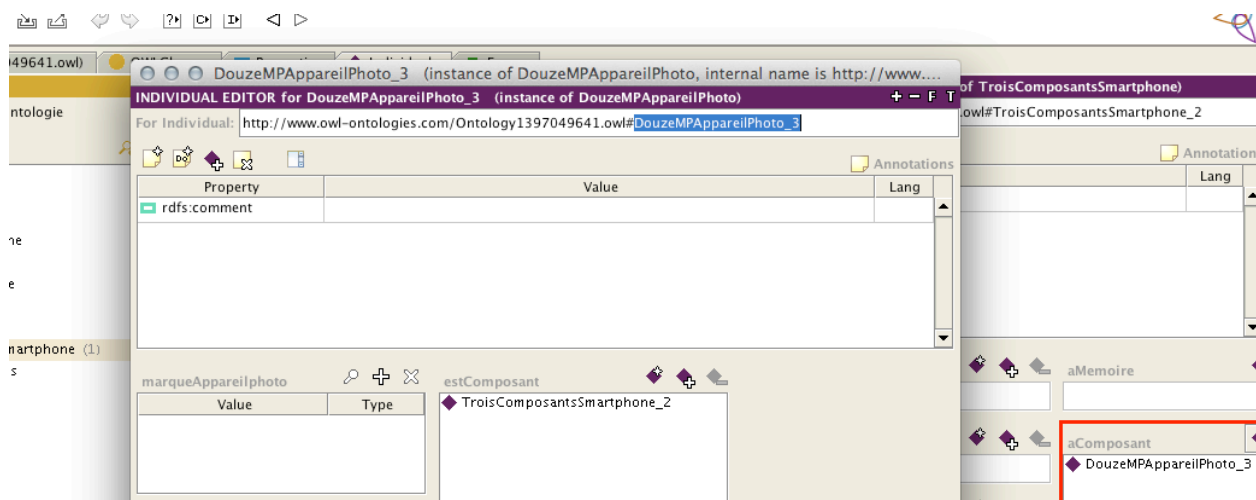


Figure 35 : Création de'une instance de DouzeMPAppareilphoto liée à une instance de Smartphone

On remarque sur la figure 35 que l'instance de mon smartphone s'est automatiquement affichée sous la propriété *estComposant*, car j'ai établi que c'était la propriété inverse de *aComposant* lors de la création des propriétés d'objets.

Maintenant, je peux choisir la marque de mon appareil photo grâce à la propriété de donnée créée précédemment. Il suffit d'appuyer sur le petit "+" et une liste de choix apparaîtra avec les couleurs disponibles. Dans le cas où il n'y a pas de proposition, on peut entrer la valeur que l'on veut selon le range défini. J'ai choisi que mon appareil photo serait un Samsung.

Value	Type
Samsung	string

Figure 36 : Sélection de la marque d'un appareil photo

Une fois que les trois composants ont été créés, la propriété *aComposant* n'est plus encadrée en rouge comme avant.

Property	Value
aAppareilphoto	
aBatterie	
aBluetooth	
aCoque	
aEtui	
aInternet	
aMemoire	
aComposant	<ul style="list-style-type: none"> DouzeMPAppareilPhoto_3 GrosseBatterie_4 TroisGInternet_5

Figure 37 : Création des instances pour les trois composants

Je vais encore créer une instance de DataSmartphone qui possède deux contraintes:

1. Il peut seulement avoir une grosse batterie s'il en a une. Je n'ai pas précisé que le composant batterie était obligatoire pour chaque smartphone.
2. Il doit avoir au moins un composant internet.

Quand je veux créer une batterie via la propriété *aBatterie*, Protégé crée intuitivement une instance de GrosseBatterie. Je crée aussi une instance TroisGInternet via la propriété

aInternet. En créant les propriétés via les sous-propriétés de *aComposant*, elles s'ajoutent automatiquement à leur super-propriété. L'instance *GrosseBatterie_16* apparaît donc sous *aBatterie* mais aussi sous *aComposant* étant donné que cette propriété est transitive. L'inverse n'est pas le cas comme on le voit sur l'image précédente.

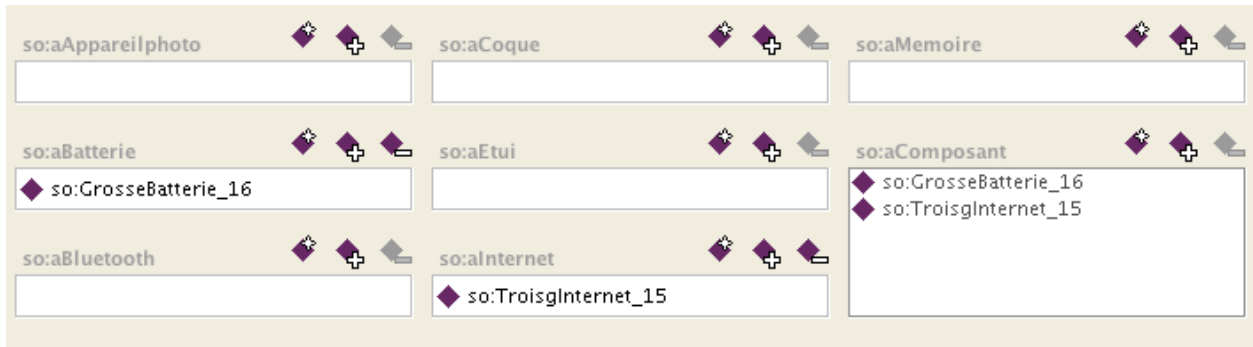


Figure 38 : Création d'une instance de DataSmartphone et de ses composants

Il est utile de rappeler que les contraintes ne fonctionnent que dans un sens. Si je crée une instance de Smartphone avec les mêmes contraintes qu'un DataSmartphone, cette instance n'appartiendra pas à la classe DataSmartphone. On peut juste être certain que toute instance d'une classe respecte ses conditions, mais on ne peut pas dire qu'une instance qui respecte ses conditions appartient à cette classe.

La figure 39 représente sur la droite deux instances de Smartphone. Une de type DataSmartphone et l'autre de type ProtegeSmartphone ainsi que leurs instances qui les composent. Sur la gauche de l'image, les classes qui dépendent de toutes les instances créées. Les rectangles jaunes représentent les classes et les violets les instances/individuels.

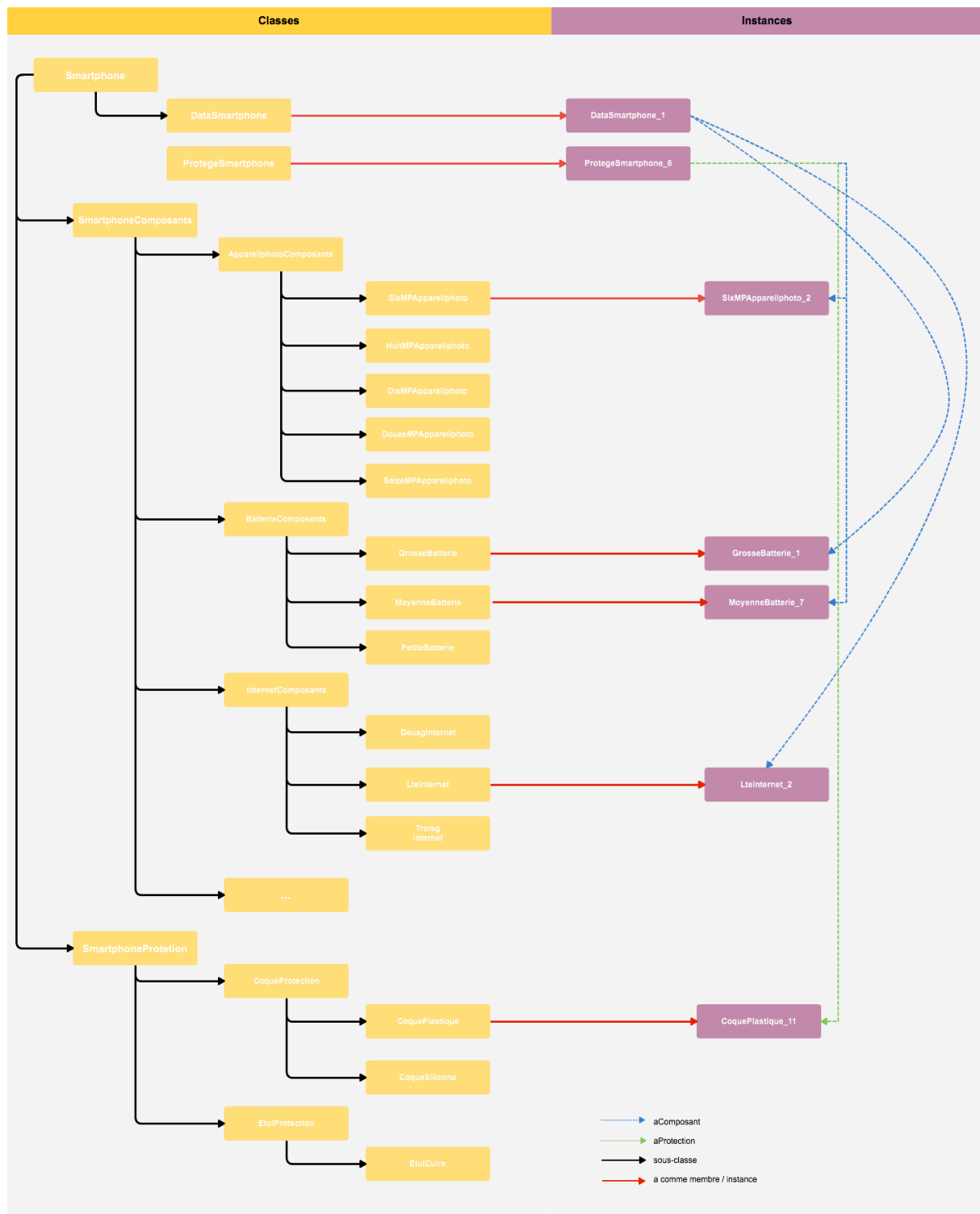


Figure 39: Extrait de la représentation graphique de l'ontologie SmartPhone

3.5.6 La consistance d'une ontologie

Le terme raisonner a été mentionné auparavant tout comme la consistance d'une ontologie durant la création d'une ontologie. Il est essentiel qu'une ontologie respecte toutes les restrictions et les règles qu'on lui a assignées. Si c'est le cas, on peut être sûr que les données que l'ontologie renvoie sont intègres.

Pour vérifier qu'une ontologie est consistante, il faut se rendre dans le menu reasoning de Protégé, sélectionner le reasoner que l'on veut (ou l'installer si on en veut un autre) et appuyer sur "check consistency". Il existe plusieurs reasoners, qui ont chacun leur petite spécificité, mais ils font le même travail. J'ai utilisé Pellet 1.5.2 pour ce travail. Selon le reasoner choisi, il va afficher un output différent pour indiquer si l'ontologie est bien consistante ou non et l'erreur ou les erreurs présentes.

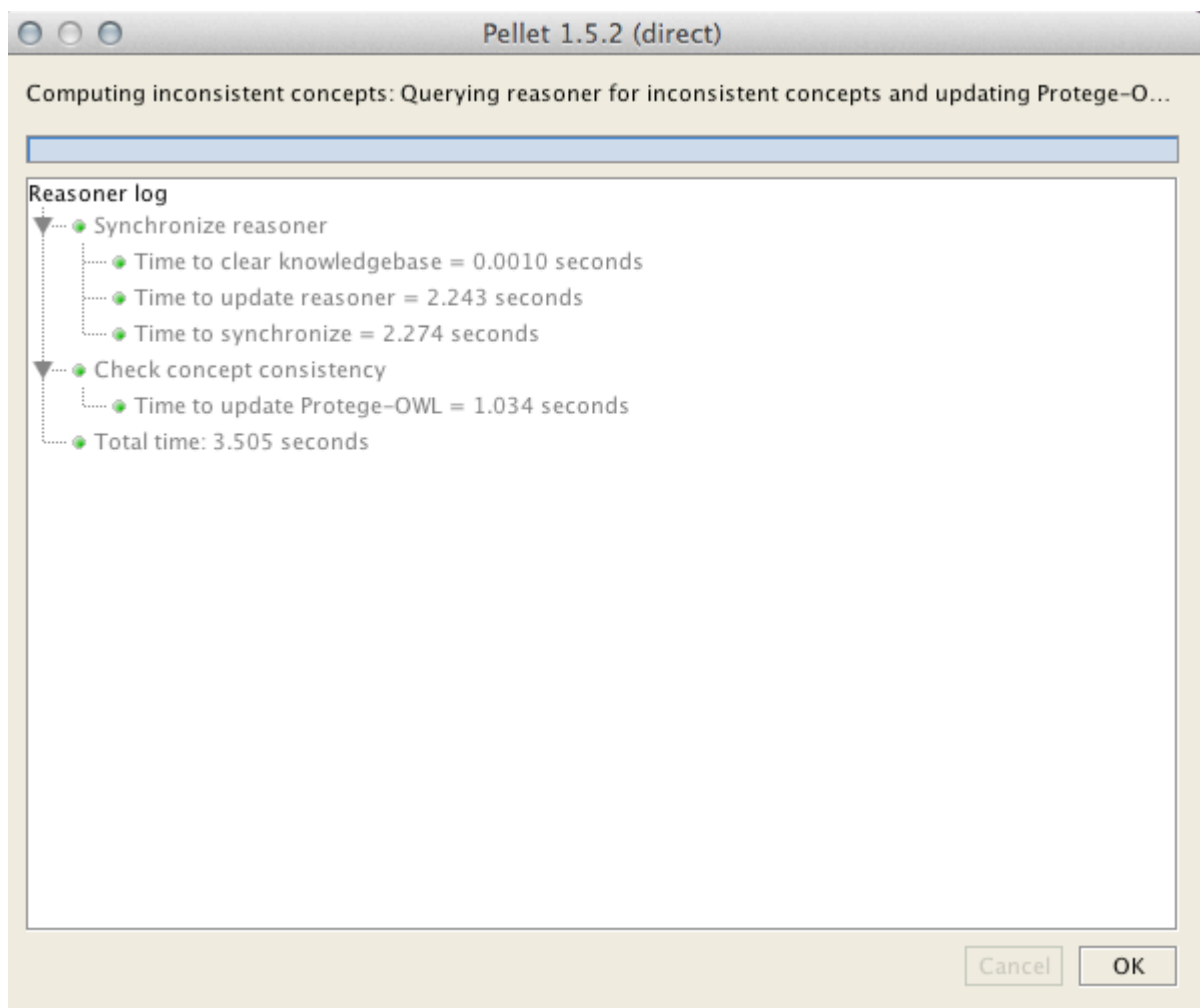


Figure 40 : Vérifier la consistance d'une ontologie grâce au reasoner

Le raisonnement permet aussi de vérifier la taxonomie ou de calculer les types inférés, c'est-à-dire toutes les restrictions, sous-classes, classes disjointes etc... grâce aux informations déjà définies dans Protégé.

Maintenant que toutes les bases théoriques concernant la création d'une ontologie ont été vues, la méthode pour l'interroger va être introduite puis détaillée. Dans le chapitre 2, les différences entre une ontologie et une base de données ont été expliquées. Même si c'est bien différent, les principes de base restent les mêmes. Il y a des données qui sont stockées et le but est de les utiliser afin d'afficher certaines informations grâce à des requêtes. Les requêtes SQL sont les plus utilisées dans les bases de données. SPARQL est le langage pour interroger des données web et donc une ontologie. Il fournit le langage d'interrogation du Web sémantique.

4

SPARQL

4.1 Définition

SPARQL [8] signifie SPARQL Protocol and RDF Query Language. C'est un langage et un protocole qui va donc permettre de faire des requêtes sur des données web et donc sur une ontologie. SPARQL est assez identique à SQL car on accède aussi aux données via ce langage. La différence est que SQL accède aux données d'une base de données et SPARQL accède aux données du Web des données, qui est "une initiative du W3C qui vise à favoriser la publication des données structurées sur le Web, non pas sous la forme de silos de données, mais en les reliant entre elles pour constituer un réseau global d'information". [2]

Ce langage de type SQL est donc utile pour interroger des données RDF comme une ontologie et permet plusieurs opérations comme SQL. Pour ce travail, seul les sélections nous intéresse car la création et modifications des données se fait avec Protégé.

Comme son nom l'indique, SPARQL travaille sur des données RDF disponibles sur Internet. RDF est toujours composé de triplets. Un triplet RDF est une association de type: (*sujet, prédicat, objet*). Le sujet est une ressource à décrire, le prédicat est comme pour tout langage de requête un type de propriété qu'on peut appliqué à cette ressource pour décrire la relation et l'objet est une autre ressource ou une donnée. On peut se représenter les données en termes de bases de données relationnelles SQL comme un tableau à trois colonnes : la colonne du sujet, la colonne du prédicat et la colonne de l'objet ou comme un modèle de graphe orienté.

Lorsque j'ai montré quelques petits exemples en OWL, j'ai parlé de caractéristiques RDF mais aussi RDFS. RDFS ou RDF Schema est aussi sous forme de triplet et fournit des éléments de base pour définir une ontologie afin de structurer des ressources. Il permet de

hiérarchiser les classes et les propriétés. RDFS va donc décrire les sous-classes par exemple. Lors de certaines requêtes SPARQL, certaines spécificités d'OWL, de RDF et de RDFS ont été utilisées pour sélectionner ce qui nous intéresse, car l'ontologie est aussi composée d'attributs de RDF, RDFS et OWL comme expliqué précédemment. Pour pouvoir utiliser toutes ces spécificités, il faut définir les espaces de noms avant la requête de sélection soit l'espace de nom de rdf, de rdfs et de l'ontologie.

1	PREFIX ont: <http://www.ontologies.com/monontologie.owl#>
2	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

Le mot clé “PREFIX” permet d'en indiquer le préfixe puis l'adresse. Devant chaque propriété utilisée dans un triplet, on doit indiquer le préfixe suivi de la propriété comme suit : « rdf:proprieteRDF » ou « ont:proprieteOntologie ».

Ensuite comme en SQL, il faut projeter les ressources choisies avec “SELECT”, puis choisir la ou les conditions avec des triplets en utilisant le mot clé “WHERE”.

C'est le minimum requis pour effectuer une requête. On peut ensuite trier les informations grâce à “ORDER BY”, rajouter des conditions avec “FILTER”, limiter l'output avec LIMIT et OFFSET etc...

Une requête est composée de triplets dans lesquelles une ou plusieurs variables sont introduites pour matérialiser des ressources, des propriétés ou des valeurs. Chaque variable doit obligatoirement commencer par un point d'interrogation et chaque triplet doit finir par un point.

Voici un exemple d'une requête basique:

```
1    PREFIX so: <http://www.owl-ontologies.com/Ontology1397049641.owl#>
2    SELECT ?smartphone ?composant WHERE {
3        ?smartphone so:aComposant ?composant.
4    }
```

- La première ligne indique l'espace nom nommé "so" de mon ontologie.
- La seconde ligne sélectionne les variables qui seront affichées.
- La troisième ligne est la clause WHERE. Le sujet est "smartphone", le prédicat est *aComposant* et l'objet est "composant". La variable *smartphone* matérialise toutes les ressources qui ont un composant et la variable *composant* représente ce composant.

Cette requête retourne donc toutes les instances qui ont un composant et affiche le nom de l'instance et de son composant. Si l'instance a plusieurs composants, l'instance sera retournée plusieurs fois avec à chaque fois le nom d'un des composants. Il est aussi possible d'exprimer des requêtes sur les représentations sous-jacentes RDF, RDFS et OWL comme expliqué précédemment. SPARQL n'est pas un langage très lisible à première vue étant donné qu'il se base sur RDF, mais une fois qu'on l'a exercé quelque peu, cela devient assez facile de comprendre les requêtes.

On va par exemple utiliser RDF pour tester le type d'une ressource grâce à « *rdf:type* ».

```
1    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2    SELECT ?subject ?object WHERE {
3        ?subject rdf:type ?object.
4    }
```

RDFS propose les prédicats *subClassOf* et *SubPropertyOf* pour tester la hiérarchie des classes.

```

1    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2    SELECT ?subject ?object WHERE {
3      ?subject rdfs:subClassOf ?object.
4    }

```

OWL propose toutes les propriétés présentées durant le chapitre 3. Il propose par exemple toutes les propriétés d'objets et de données qu'on a créées, les restrictions et les propriétés natives d'OWL comme les classes, les classes disjointes, etc...

Tout comme SQL, SPARQL possède les opérations de comparaison, arithmétiques et booléennes habituelles et plusieurs agrégats comme GROUP BY, ORDER BY, MIN, MAX, COUNT, AVG, etc... [9] ainsi que plusieurs fonctions comme RAND, CEIL, YEAR, etc... SPARQL propose aussi des opérateurs tels que l'union, le filtre, la jointure, etc...

4.2 SPARQL dans Protégé : exemples de requêtes

Il est possible d'écrire et de tester ses requêtes SPARQL directement dans Protégé. Pour cela, il faut ouvrir le panel dans le menu "Reasoning" et sélectionner « Open SPARQL Query panel ».

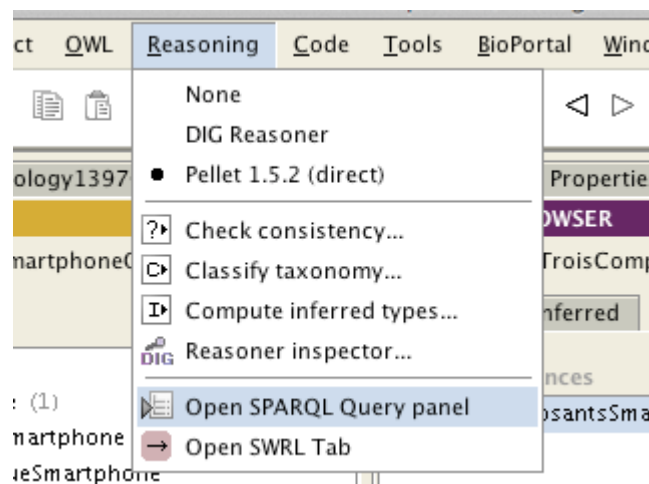


Figure 41 : Ouvrir le panel SPARQL dans Protégé

Une fois le panel SPARQL ouvert, il suffit d'entrer une requête et le résultat s'affichera sur la droite sous forme de tableau. A noter que dans Protégé il n'est pas nécessaire d'indiquer les

espaces de nom. Ils y sont déjà automatiquement à la création de l'ontologie comme on peut le voir dans l'onglet "Metadata". Il ne faut pas tout de même oublier de préciser le préfixe de l'ontologie si elle n'en a pas encore. On peut également ajouter un espace de nom dans le cas où il en manque un.

Namespace Prefixes	
Prefix	Namespace
owl	http://www.w3.org/2002/07/owl#
protege	http://protege.stanford.edu/plugins/owl/protege#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
so	http://www.owl-ontologies.com/Ontology1397049641.owl#
swrl	http://www.w3.org/2003/11/swrl#
swrlb	http://www.w3.org/2003/11/swrlb#

Figure 42 : Déclaration des espaces de noms

Voici le résultat que Protégé affiche après l'exécution d'une requête basique où on veut afficher une classe et sa superclasse correspondante.

Query	
<pre>SELECT ?subject ?object WHERE { ?subject rdfs:subClassOf ?object }</pre>	
Execute Query	
Results	
subject	object
DixMPAppareilphoto	AppareilphotoComposant
BluetoothOne	BluetoothComposant
PetiteMemoire	MemoireComposant
MemoireComposant	SmartphoneComposants
MoyenneBatterie	BatterieComposant
LteInternet	InternetComposant

Figure 43 : Affichage d'un résultat d'une requête SPARQL dans Protégé

Si on veut afficher toutes les propriétés de donnée, il faut utiliser le prédicat owl:DatatypeProperty. On veut afficher toutes les ressources qui sont de type datatype.

Query	
<pre>SELECT ?datatype WHERE { ?datatype rdf:type owl:DatatypeProperty }</pre>	
Execute Query	
Results	
datatype	
marqueAppareilphoto	
marqueBluetooth	
marqueInternet	
marqueBatterie	
marqueMemoire	
couleur	

Figure 44 : Affichage de toutes les propriétés de données de mon ontologie

Revenons à la première requête expliquée de ce chapitre. Je veux connaître toutes les entités de type Smartphone et leur(s) composant(s), il faut d'abord déclarer que les entités sont de type Smartphone et ensuite utiliser la propriété d'objet *aComposant* comme prédicat. Quand

on déclare plusieurs triplets, il est obligatoire de terminer chaque triplet par un point.

```

1      SELECT ?entity ?composant WHERE {
2      ?entity rdf:type ?so:Smartphone.
3      ?entity so:aComposant ?composant.
4      }

```

A noter qu'on peut simplement utiliser un "a" à la place du prédicat `rdf:type` et que le point-virgule à la fin d'un triplet permet de lister des triplets qui partagent le même sujet.

```

1      SELECT ?entity ?composant WHERE {
2      ?entity a so:Smartphone;
3      so:aComposant ?composant.
4      }

```

Ces deux requêtes retournent exactement la même chose soit une entité `Smartphone_9` qui a un composant internet 3G et une grosse batterie et une entité `Smartphone_13` qui a un appareil photo de 10MP, une mémoire moyenne et un bluetooth 2, ...

Results	
smartphone	composant
◆ so:Smartphone_9	◆ so:TroisgInternet_11
◆ so:Smartphone_9	◆ so:GrosseBatterie_12
◆ so:Smartphone_13	◆ so:DixMPAppareilphoto_6
◆ so:Smartphone_13	◆ so:MoyenneMemoire_7
◆ so:Smartphone_13	◆ so:BluetoothTwo_8
◆ so:TroisComposantsSmartphone_2	◆ so:DouzeMPAppareilPhoto_3

Figure 45 : Affichage de toutes les instances de Smartphone et leur(s) composant(s)

Si une instance de Smartphone ne possède pas de composant, elle ne va pas s'afficher. Prenons l'exemple où on veut afficher toutes les instances et leurs composants s'ils en ont. Pour cela il faut définir une clause optionnelle en utilisant le mot-clé `OPTIONAL` puis définir ensuite le(s) triplet(s) concerné(s).

```
1    SELECT ?entity ?composant WHERE {
2    ?entity a so:Smartphone.
3    OPTIONAL {?entity so:aComposant ?composant.}
4    }
```

On peut également classer les résultats par les noms d'instances grâce à l'agrégat ORDER BY et en afficher uniquement trois avec LIMIT. Ces deux agrégats doivent se trouver à l'extérieur de la clause WHERE.

```
1    SELECT ?entity ?composant WHERE {
2    ?entity a so:Smartphone.
3    OPTIONAL {?entity so:aComposant ?composant.}
4    }
5    ORDER BY ?entity
6    LIMIT 3
```

Une dernière possibilité importante est de filtrer les résultats grâce à l'agrégat FILTER. Imaginons que je veux maintenant connaître le smartphone qui possède l'instance GrosseBatterie_13

```
1    SELECT ?entity ?composant WHERE {
2    ?entity so:aComposant ?composant.
3    FILTER (?composant = so:GrosseBatterie_13).}
```

FILTER propose plusieurs opérateurs, tests et accesseurs, pour filtrer précisément les données que vous pouvez connaître en visitant le site Web de la ressource [10].

5

La Smart Home

5.1 Présentation du projet et du scénario

Le but de ce travail de Bachelor a été de travailler avec un doctorant, Mr Abdaladhem Albreshne du groupe Génie Logiciel du DIUF afin de proposer une architecture logicielle pour gérer un environnement résidentiel intelligent comme une maison intelligente ou smart home en anglais. Cette approche est basée sur une description sémantique et sur les ontologies pour découvrir et composer des services dans une maison où l'environnement serait changeant. L'objectif est donc de fournir et de contrôler une maison dite "intelligente", de définir un environnement spécifique pour offrir aux utilisateurs un assistant afin d'obtenir des solutions optimisées pour composer des services web. Une smart home est composée de différents objets dit "intelligents", ce qui signifie qu'ils sont associés à des actuateurs et à des senseurs qui fournissent leurs prestations via des services web. L'environnement est aussi bien décrit de manière syntaxique que sémantique grâce à l'ontologie basée sur OWL. Il est possible de piloter différents scénarios écrits avec un langage de gestion de processus nommé GPL4SE [11] développé par le groupe génie logiciel. Le WSDL est utilisé pour décrire les services web.

Je me suis basé sur un scénario assez simple décidé par le groupe Génie Logiciel pour construire une ontologie. La maison possède trois pièces. La première possède une porte d'entrée, une lampe, un chauffage (radiateur), une fenêtre et différents objets pour la cuisine comme des plaques de cuisson, un évier, etc... La seconde pièce est la chambre à coucher. Il y a donc un lit, une lampe, un chauffage et une fenêtre. La troisième et dernière pièce est la salle de bain. Elle possède également une lampe, une fenêtre et un chauffage ainsi que des toilettes, une douche et un lavabo. Une partie de ces objets sont intelligents et possèdent des actuateurs et/ou des senseurs.

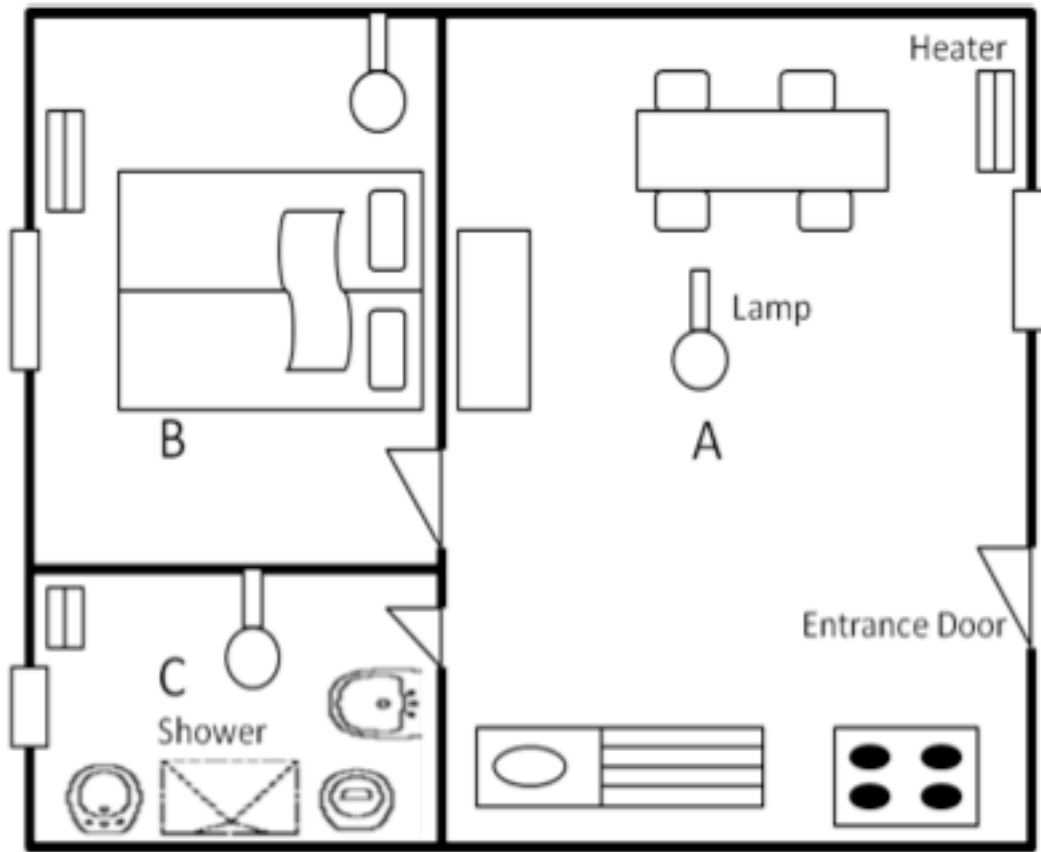


Figure 46 : Schéma de la maison intelligente (pris dans [11])

Durant ce travail de Bachelor, je devais comprendre ce scénario, mais aussi l'étendre. Je suis donc resté sur une maison de trois pièces, mais j'ai créé beaucoup plus d'objets intelligents dans l'ontologie et donc plus d'actuateurs, plus de senseurs, plus d'actions, plus de requêtes et plus de propriétés. Le but est de pouvoir offrir un scénario plus complexe tout en utilisant la même ontologie. De mon côté ça me permettait d'accroître de nouvelles connaissances en OWL et dans la création d'ontologies.

Le prochain sous-chapitre présente l'ontologie de cette maison intelligente et le suivant l'architecture du framework qui permet de découvrir et d'utiliser de manière efficiente les services fournis par l'environnement intelligent et les requêtes SPARQL utilisées et leur rôle.

5.2 L'ontologie de la Smart Home

Comme dit précédemment, la maison possède des objets qui peuvent être intelligents. Un objet est considéré comme intelligent s'il a au moins un actuateur ou un senseur.

Mon ontologie a été construite selon plusieurs concepts :

- 1) Les entités qui représentent les objets, les personnes et les pièces de la maison. Dans ce projet, les personnes ne sont pas utilisées mais pourraient l'être dans le futur.
- 2) Les actuateurs qui vont permettre aux entités de réaliser certaines actions.
- 3) Les senseurs qui vont permettre de récolter des informations sur une entité.
- 4) Les publishers qui vont publier les événements des senseurs.
- 5) Les services qui représentent les actions des actuateurs, les requêtes des senseurs et les événements des publishers.

Une entité peut avoir un senseur ou/et un actuateur. Chaque senseur peut posséder un ou plusieurs publishers qui vont publier un événement (event) et une ou plusieurs requêtes (query). Chaque actuateur est associé à une ou plusieurs actions. Chaque pièce (location) possède une ou plusieurs entités.

La figure 47 montre le schéma de l'ontologie de domaine qui représente ces différents concepts.

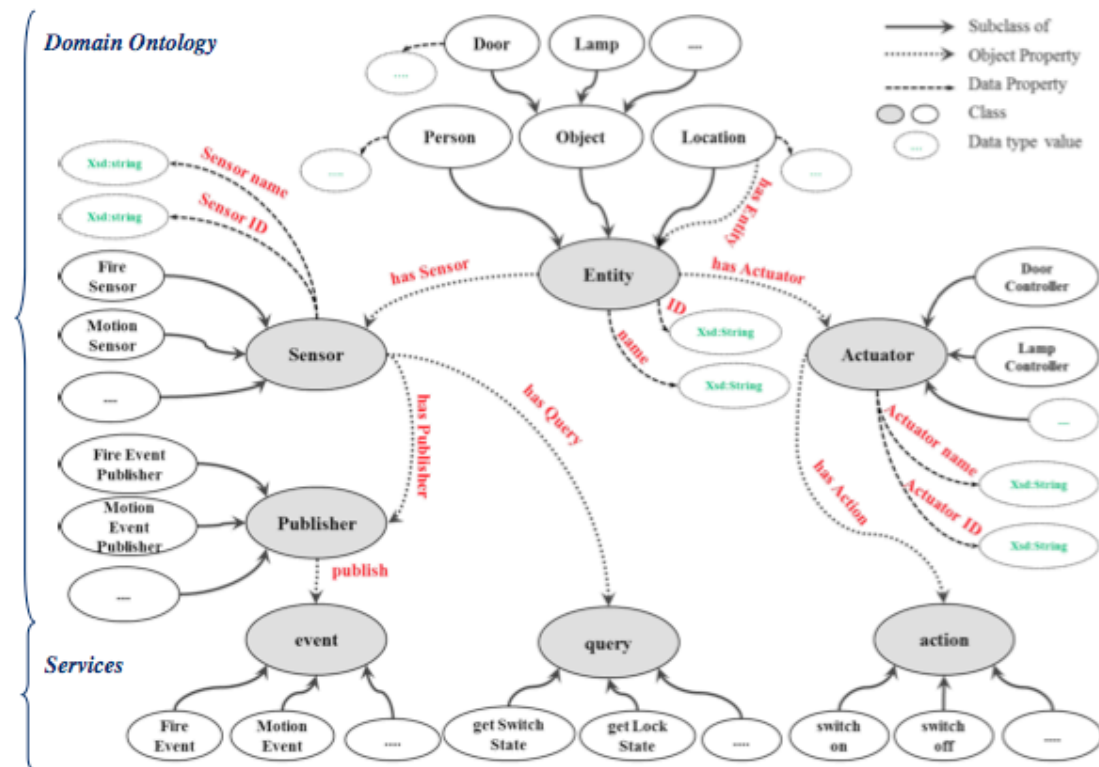


Figure 47 : Représentation schématique de l'ontologie de la smart home (pris dans [11])

5.2.1 Les classes

Chaque concept est représenté par une classe comme pour l'ontologie Smartphone présentée dans la partie théorique. Une hiérarchie a aussi été respectée. On retrouve tous les concepts mentionnés auparavant.

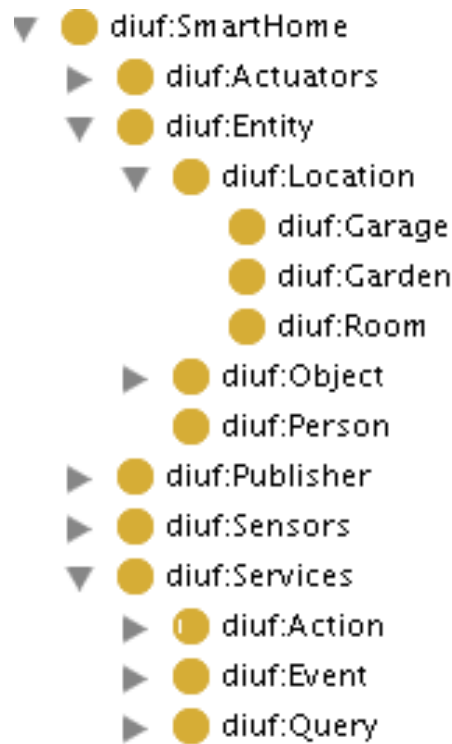


Figure 48 : Les classes de l'ontologie de la smart home

Comme on le voit sur la figure 48, l'ontologie est hiérarchisée selon les concepts. On retrouve donc :

- la classe Actuators qui contient les différents types d'actuateurs (sous-classes). Chaque type d'actuateur correspond à une classe.
- la classe Entity qui contient des lieux (Location), des objets (Objects) et des personnes (Person). Dans le scénario, j'ai uniquement utilisé la classe Objects qui contient tous les types/classes d'objets que peut contenir une maison et la classe Room qui contient les pièces de la maison.
- la classe Publisher qui contient les différents types de Publisher.
- la classe Sensors qui contient les différents types de senseurs (sous-classes).
- la classe Services qui a comme sous-classe:
 - la classe Action qui contient les différentes actions qu'un actuateur peut faire.
 - la classe Event qui contient les types d'événements.
 - la classe Query qui contient les types de requêtes qu'un senseur peut effectuer.

Les objets:

La classe Objet possède beaucoup de sous-classes. La hiérarchie des objets a été pensée dès le début du travail. En effet, une ontologie est d'abord une façon de penser une structure, une hiérarchie. Pour cette ontologie, j'ai décidé de classer les objets en catégories. La classe Construction contient les éléments qui font partie de la structure d'une maison, ses bases. Ensuite, j'ai classé les objets d'après leur fonctionnalité et j'ai créé une catégorisation comme Migros et Jumbo le fait par exemple. La classe Furniture contient les meubles qui sont nécessaires à un salon ou à une chambre par exemple. La classe Electric contient les objets qui servent à éclairer la maison. Avec cette classification, une pièce possédera normalement beaucoup d'objets de la même classe. Certaines classes par contre seront utilisées dans toutes les pièces comme la classe Electric. Il est possible de créer une multitude de classes et d'objets car tout objet peut être intelligent. On peut penser aux objets de la cuisine comme un four, un mixeur, une machine à café, des plaques de cuisson, un frigo ainsi qu'aux objets d'une salle de bain comme une douche, un évier, une machine à laver, un séchoir, des toilettes et aussi aux objets d'un salon et d'une chambre à coucher comme une télévision, un lit, une chaise ou encore les objets rudimentaires comme un radiateur, une lampe, une fenêtre, une porte, etc... Les caméras pour la sécurité ainsi que des alarmes pour le feu et le gaz sont de bons exemples d'objets intelligents qui pourraient être bien exploités dans divers scénarios.

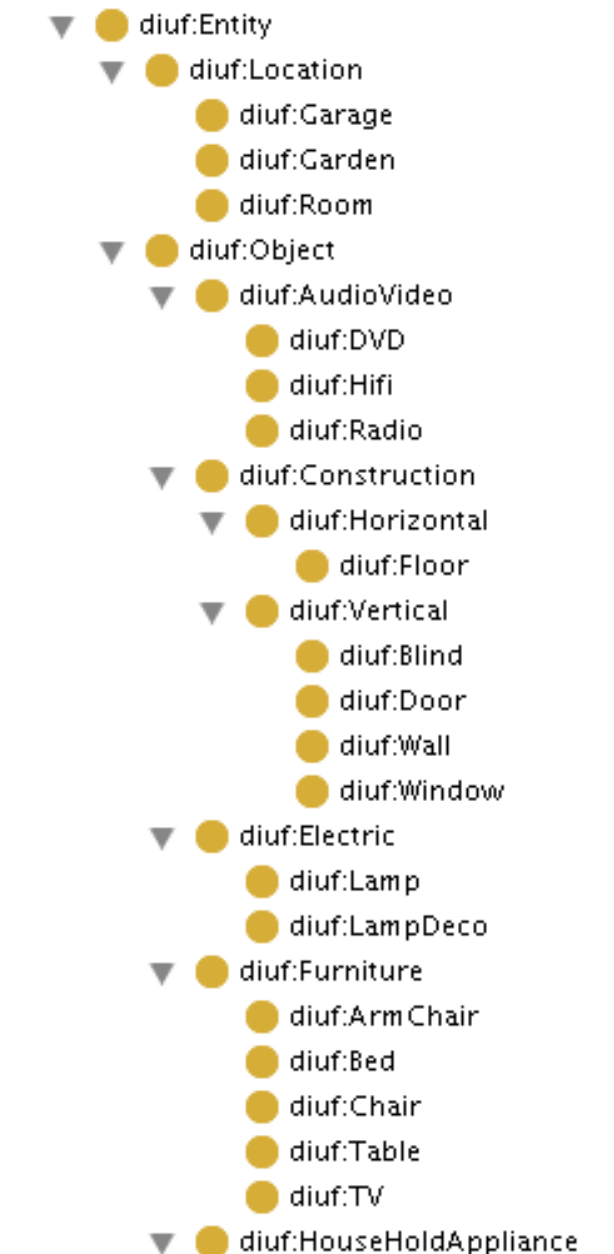


Figure 49 : Hiérarchie des classes pour les objets

Les actuateurs:

La classe Actuators possède une sous-classe nommée Control qui elle-même possède plein de sous-classes qui représentent les actuateurs. Il y a par exemple la classe AccessController qui va posséder des instances qui vont contrôler l'accès à certaines entités comme les pièces par exemple ou la classe CooktopController qui contiendra des instances pour contrôler les plaques de cuisson de la cuisine. Plusieurs entités peuvent utiliser le même type d'actuateur si par exemple deux entités offrent exactement les mêmes actions. Il n'y a donc pas forcément le même nombre de classe d'actuateurs que d'objets.

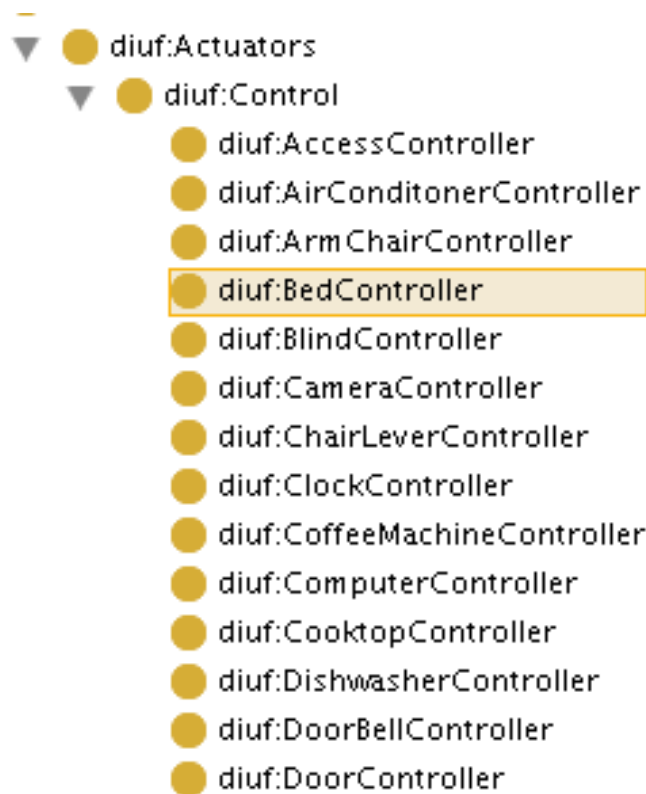


Figure 50 : Hiérarchie des classes pour les actuateurs

Les senseurs:

La classe Sensors possède aussi plusieurs sous-classes car il y a plusieurs types de senseurs. Par exemple, la classe DoorSensor contiendra des instances de senseurs qui seront liées à des portes tandis que la classe SwitchSensor aura des instances qui seront liées à des objets qui peuvent être allumés (switchOn) ou éteints (switchOff).

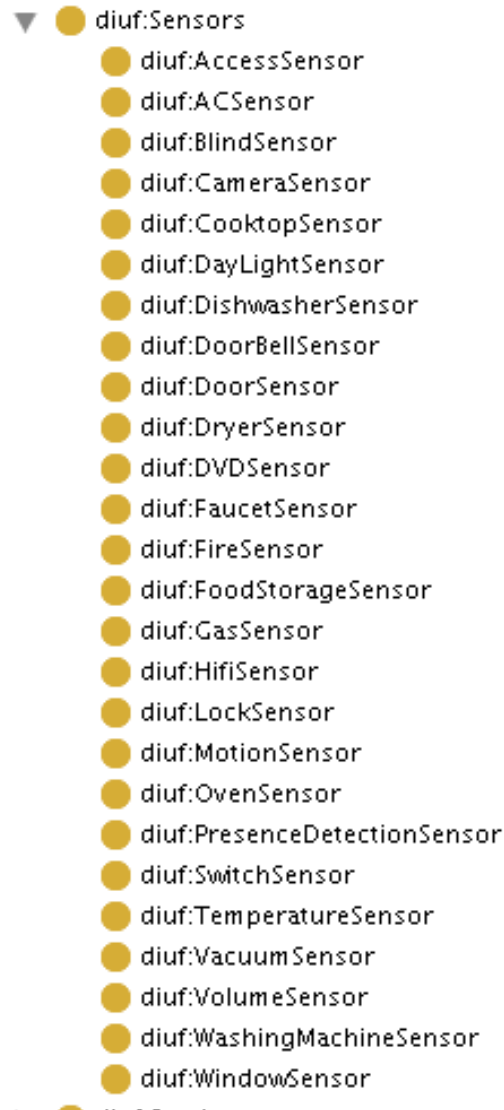


Figure 51 : Hiérarchie des classes pour les senseurs

Les publishers:

Chaque senseur peut avoir un ou plusieurs publishers qui lui sont affiliés. Ils permettent d’afficher des informations sur un événement. A chaque publisher doit donc correspondre un ou plusieurs événements. La classe CameraEventPublisher publie donc des informations sur un ou des événements liés aux caméras de la maison.

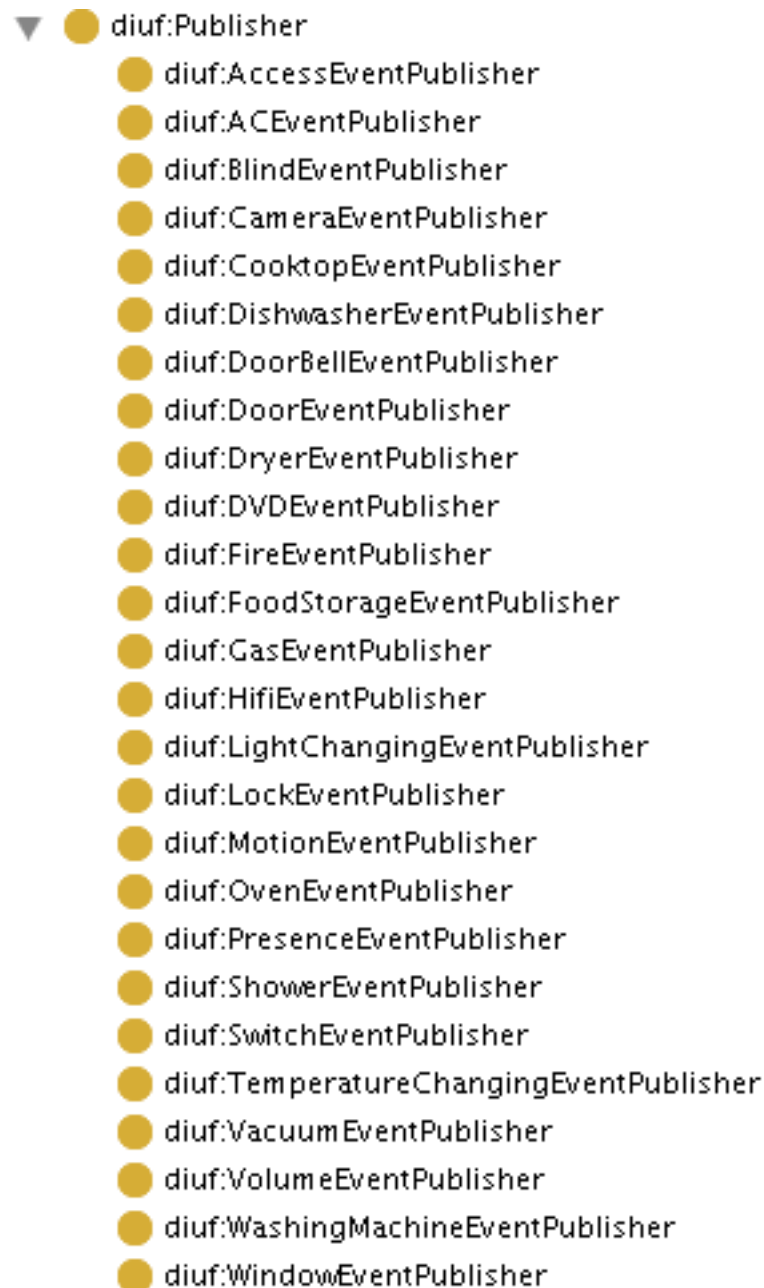


Figure 52 : Hiérarchie des classes pour les publishers

Les actions:

Chaque actuateur peut effectuer différentes actions. Un actuateur peut par exemple augmenter ou diminuer la température d'une pièce ou du chauffage, éteindre ou allumer un objet que ça soit la télévision, une caméra ou la lumière, un autre va contrôler le robinet en ajustant la température de l'eau et en l'ouvrant ou le fermant. Il y a donc beaucoup d'actions possibles, il suffit d'être imaginatif pour ce qu'on veut proposer.

La figure 53 montre un aperçu des classes représentant les types d'actions.

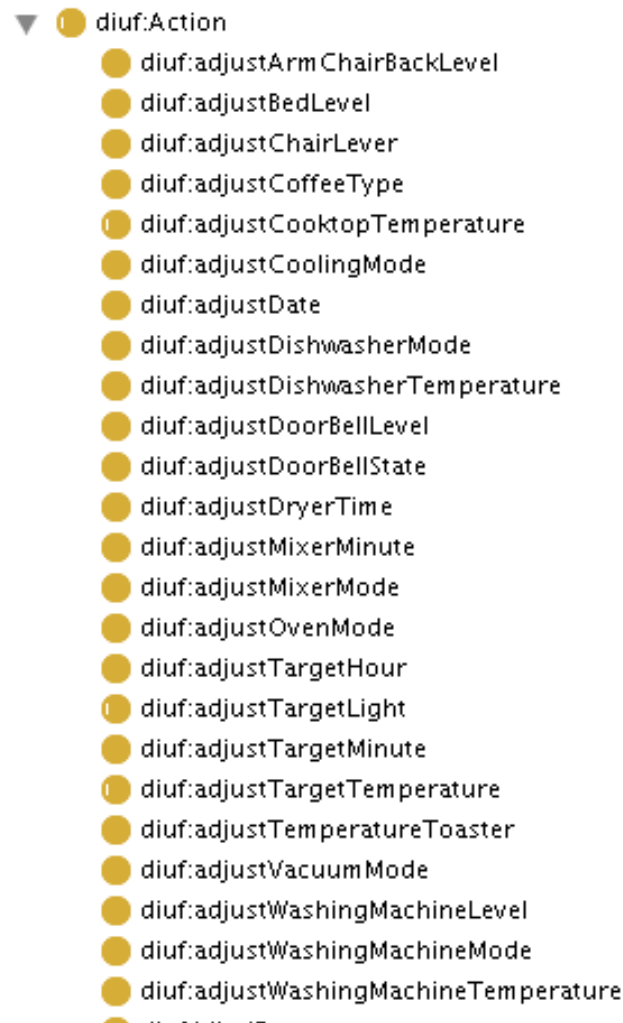


Figure 53 : Hiérarchie des classes pour les actions

Les requêtes:

Chaque senseur peut être lié à une ou plusieurs requêtes. Le senseur d'une porte pourra nous informer si elle est ouverte ou fermée, celui de la télévision pourra nous informer sur son état (allumé ou éteint) et son volume. Un autre nous renseigne sur la température d'une pièce ou de l'eau de la douche. Là aussi, beaucoup de requêtes sont possibles. La figure 54 montre celles qui ont été créées.

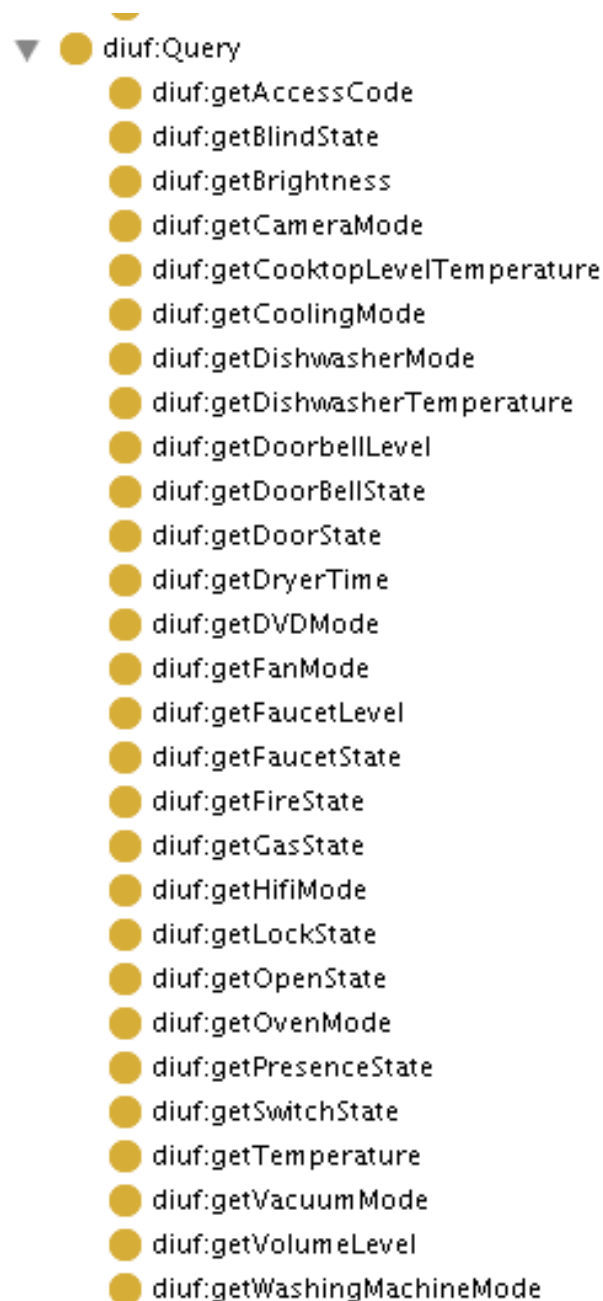


Figure 54 : Hiérarchie des classes pour les requêtes

Les événements:

Un événement se déclenche quand une requête est effectuée. Il y a plusieurs types d'événements. Je l'ai classé par rapport à leur publisher. La classe CameraEvent contient des instances d'événements de caméras. Un événement serait par exemple la demande d'éteindre une caméra.



Figure 55: Hiérarchie des classes pour les événements

5.2.2 Les propriétés d'objets

Comme le schéma de l'ontologie de domaine le présentait bien, il y a quatorze propriétés d'objets.

1. *hasActuator* et son inverse *isActuatorOf* pour lier un objet et son ou ses actuateurs
2. *hasSensor* et son inverse *isSensorOf* pour lier une entité et son ou ses senseurs

3. *hasPublisher* et son inverse *isPublisherOf* pour lier un senseur et son publisher
4. *hasAction* et son inverse *isActionOf* pour lier un actuateur et son ou ses actions
5. *hasQuery* et son inverse *isQueryOf* pour lier un senseur et sa ou ses requêtes
6. *hasEvent* et son inverse *isEventOf* pour lier un publisher et son événement
7. *hasEntity* et son inverse *hasLocation* pour lier un objet et une pièce

Les domaines et les ranges sont facilement déductibles. Le domaine de la propriété *hasActuator* est la classe Entity et le range la classe Actuators. Comme expliqué dans la partie théorique, c'est l'inverse pour la propriété inverse. Pour la propriété *hasAction*, le domaine est la classe Actuators et le domaine la classe Action.

Un actuateur ne peut être attaché qu'à un objet. En effet, une pièce ne peut pas avoir d'actuateur. Par contre, une pièce peut avoir un senseur. A la place de créer une instance de Floor dans chaque pièce, une instance d'un senseur MotionSensor est attachée à une pièce (Room) qui permettra de savoir si quelqu'un est dans une pièce ou non. La figure 56 montre toutes les propriétés d'objets de cette ontologie.

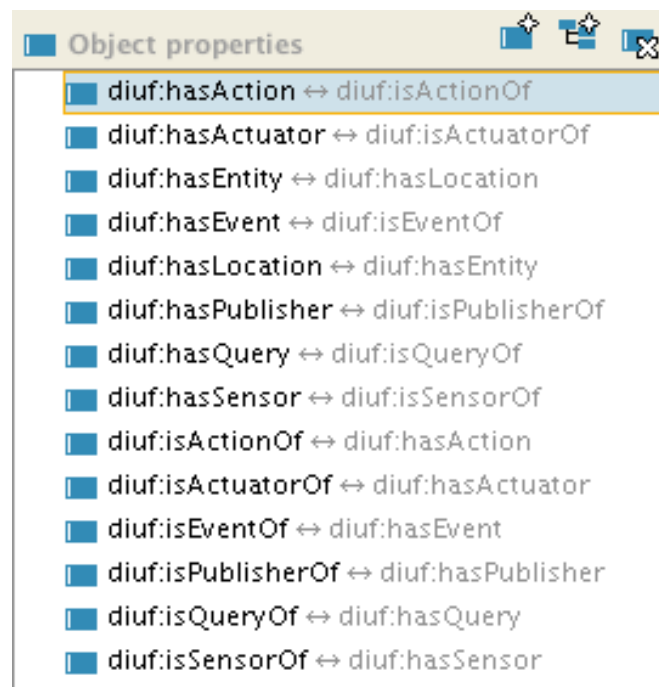


Figure 56 : Les propriétés d'objets de la smart home

Il est très important qu'une instance d'un objet ne puisse être que dans une seule pièce. Si

l'instance `Door_22` est dans la salle de bain, elle ne peut pas être dans la chambre à coucher. Pour que cette condition soit remplie, il faut que la propriété *hasLocation* soit fonctionnelle et que son inverse *hasEntity* soit inversement fonctionnelle. En effet, chaque objet ne peut appartenir qu'à une seule pièce de la maison.

C'est en fait la même chose pour chaque propriété d'objet. Toute instance d'actuateur ne peut appartenir qu'à un seul objet. Une instance de senseur ne peut être lié qu'à une seule instance d'une entité, etc...

Les propriétés *isActionOf*, *isActuatorOf*, *isEventOf*, *isPublisherOf*, *isQueryOf*, *hasLocation* et *isSensorOf* sont donc fonctionnelles tandis que leur propriété inverse est inversement fonctionnelle.

5.2.3 Les restrictions

Comme expliqué auparavant, il y a quelques relations et restrictions à indiquer. Chaque entité peut avoir zéro, un ou plusieurs actuateurs et il en est de même pour les senseurs, mais chaque type d'entité ne peut avoir que certains types d'actuateurs et de senseurs. Pour cette ontologie, Seul des restrictions universelles ont été employées. L'entité `Lamp` par exemple peut avoir uniquement un actuateur de type `LampController` et un senseur de type `DayLightSensor`. Les propriétés *hasActuator* et *hasSensor* et une restriction universelle (*only*) sont utilisées pour définir ceci.

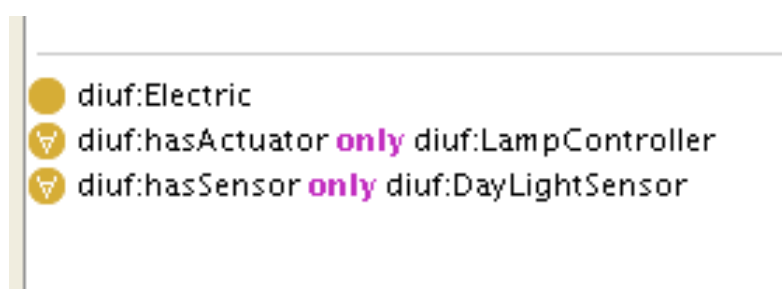


Figure 57 : Restrictions pour les instances de la classe `Lamp`

Un actuateur de type `LampController` ne fournit que trois actions soit le fait d'allumer la lampe, de l'éteindre et de régler sa luminosité. Ces actions sont nommées *switchOn*, *switchOff* et *adjustTargetLigth*. La propriété *hasAction* permet de lier un actuateur à son ou ses actions.

- diuf:Control
- ▽ diuf:hasAction **only** (diuf:switchOff **or** diuf:switchOn **or** diuf:adjustTargetLight)

Figure 58 : Restriction pour les instances de la classe LampControllor

Un capteur peut posséder aucun, un ou plusieurs publishers et une ou plusieurs requêtes selon la définition de la classe à laquelle il appartient. Un capteur de type DayLightSensor peut avoir seulement un publisher de type LightChangingEventPublisher et deux requêtes: getSwitchState et getBrightness.

- diuf:Sensors
- ▽ diuf:hasPublisher **only** diuf:LightChangingEventPublisher
- ▽ diuf:hasQuery **only** (diuf:getSwitchState **or** diuf:getBrightness)

Figure 59 : Restrictions pour les instances de la classe DayLightSensor

Le publisher possède un événement. Le publisher LightChangingEventPublisher ne peut avoir qu'un événement de type LightChangingEvent.

- diuf:Publisher
- ▽ diuf:hasEvent **only** diuf:LightChangingEvent

Figure 60 : Restriction pour les instances de la classe LightChangingEventPublisher

Cette construction est identique pour chaque type d'entité présente dans l'ontologie que ça soit une porte, une fenêtre, etc...

5.2.4 Les propriétés de données

Dans cette ontologie, il y a beaucoup de propriétés de données. Le but est de contrôler une maison dite “intelligente” et beaucoup d’informations sur chaque instance sont nécessaires. Le premier est de donner un identificateur et un nom à chaque instance que ça soit une pièce, un objet, un actuateur, etc... Ces informations seront employées dans le framework pour insérer les données et les découvrir. Il faut que chaque instance ait un nom ou un identifiant unique afin de ne pas insérer plusieurs fois les mêmes instances.

- une entité a une propriété `entityID` et `entityName`
- un actuateur a une propriété `actuatorID` et `actuatorName`
- un senseur a une propriété `sensorID` et `sensorName`.
- une pièce (Room) a une propriété `entityID`, `entityName`, `locationID` et `locationName`.

Chaque objet intelligent fournit via son ou ses senseurs des informations qui sont accessibles via les requêtes. Si on veut connaître l’état de la télévision ou de la lampe, le senseur doit connaître les données de son objet. Ces données sont les propriétés de données. Une lampe a comme donnée par exemple son état (`switchState`) et sa luminosité (`brightness`). La télévision elle nous indique son niveau de volume (`volumeLevel`), son état (`switchState`). Pour une douche, on veut savoir si le robinet est ouvert ou non (`faucetState`) et connaître la température de l’eau (`faucetLevel`). Pour cette dernière donnée, j’ai décidé de ne pas connaître la température exacte, mais juste un niveau: froid, tiède ou chaud. La température de la pièce pourra être connue en degré, ce sera donc une autre propriété de données. `faucetLevel` aura comme range un string et `temperature` un float.

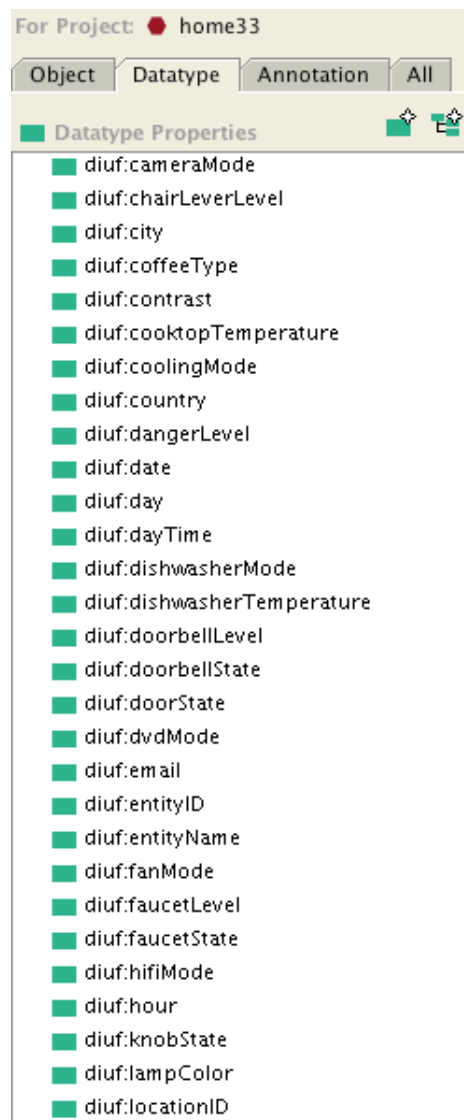


Figure 61 : Les propriétés de données pour la smart home

En plus de toutes ces propriétés, deux sont très importantes pour ce projet: l'adresse OWL-S [12] et l'adresse WSDL [13]. Chaque action et requête doivent posséder ces deux informations. Les web services sont fournis et décrit grâce à leur fichier WSDL et OWL-S.

Le WSDL (Web Services Description Language) et OWL-S (Web Ontology Language for Web Services) permettent de décrire un service web en se basant sur le XML et sont en relation. Ils vont utiliser des messages SOAP pour communiquer à travers un réseau. Le service est fournit par une interface WSDL et propose une description OWL-S pour le découvrir et comprendre ses fonctions.

Le WSDL permet de spécifier un protocole et un mécanisme d'encodage pour un service web

afin de décrire les moyens d'interagir avec les autres services offerts.

Chaque service web doit avoir un fichier WSDL disponible sur le web qui le décrit et permet de savoir avec quels autres services il peut interagir. Dans l'ontologie, le datatype "wsdlLocation" permet d'indiquer le lien URI vers le fichier WSDL pour chaque service.

Le OWL-S décrit lui ce que fait un service web, ses limitations et ses exigences. Il va donc décrire les différentes instances des actions et des requêtes de l'ontologie. OWL-S nécessite une description supplémentaire pour une spécification plus complète du service et WSDL est le plus couramment utilisé. Les fichiers WSDL et OWL-S doivent être disponibles par le framework que ça soit sur un serveur ou un réseau.

Après la création de toutes les propriétés de données, on peut définir le domaine et le range de chaque propriété comme mentionné dans la partie théorique. Dans cette ontologie, le domaine est plus compliqué à bien définir, car une propriété touche souvent plusieurs classes: l'entité et ses services (action, requête et événement). Par exemple, la propriété "brightness" qui permet d'indiquer l'état de la luminosité d'une lampe a plusieurs classes comme domaine. En effet, cette propriété doit être accessible par :

- la classe Lamp pour indiquer l'intensité lumineuse de chaque instance de Lamp
- l'action "adjustTargetLight" qui utilise cette propriété comme input
- la requête "getBrightness" qui utilise cette propriété comme output
- l'événement LightChangingEvent qui utilise cette propriété pour l'afficher comme information lorsqu'une requête a été faite

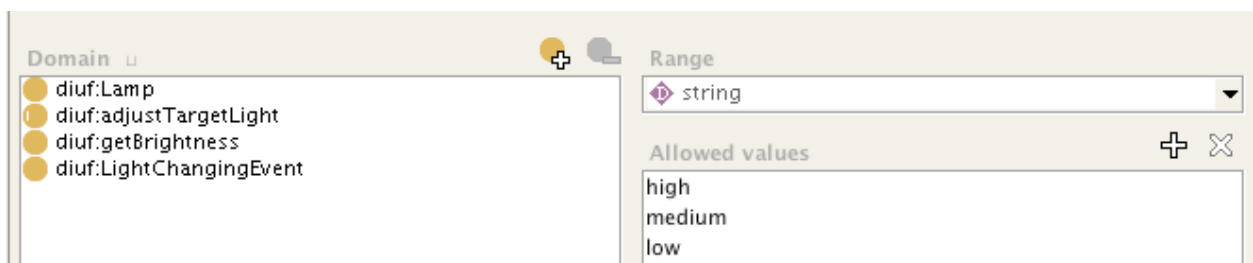


Figure 62 : Spécification du domaine et du range de la propriété brightness

Pour rappel, une entité n'est pas seulement un objet, mais peut être aussi une pièce de la maison. La classe Location a comme restriction qu'elle peut être liée à plusieurs senseurs

mais seulement un capteur pour le feu, pour le mouvement, pour la présence et pour la température. Étant donné que la classe Room hérite de la classe Location, elle possède aussi cette restriction comme on le remarque sur l'image ci-dessous. La 3ème ligne est plus claire car c'est justement cette restriction héritée.

		NECESSARY & SUFFICIENT	
● diuf:Location		NECESSARY	<input type="checkbox"/>
● diuf:hasEntity only diuf:Entity			<input type="checkbox"/>
● diuf:hasSensor only (diuf:FireSensor or diuf:PresenceDetectionSensor or diuf:TemperatureSensor or diuf:MotionSensor)		INHERITED	<input type="checkbox"/>
		[from diuf:Location]	<input type="checkbox"/>

Figure 63 : Restriction pour les instances de la classe Location

5.2.5 L'instanciation de la maison

Pour créer cette maison intelligente, il suffit de créer les instances dans Protégé comme expliqué dans la partie théorique. Étant donné que l'ontologie est très restrictive (beaucoup de restrictions universelles), il n'est possible d'attacher un actuateur de type DoorController à une instance de Lamp par exemple. Lors de la création d'une instance d'un objet, il est seulement possible de lui attacher un ou deux actuateurs définis dans les restrictions et c'est la même chose pour les actions d'un actuateur, les capteurs, les requêtes, etc...

Même s'il existe beaucoup de types d'objets qui pourront être intelligents pour pouvoir ensuite étendre la maison avec ceux-ci, je me suis concentré sur le scénario de base expliqué en début de chapitre. La maison est donc composée de trois pièces qui possèdent chacune certains objets.

Exemple d'une pièce de la SmartHome: le hall d'entrée

On retrouve toutes les propriétés d'objets et de données affiliées à une pièce. Elle a comme propriétés de donnée un entityID, un entityName, une locationID et une locationName. Pour celles d'objets, on retrouve la propriété hasEntity qui comprend les instances, donc les objets intelligents de cette pièce, et la propriété hasSensor qui contient un capteur qui détecte la présence de quelqu'un et un second pour la température.

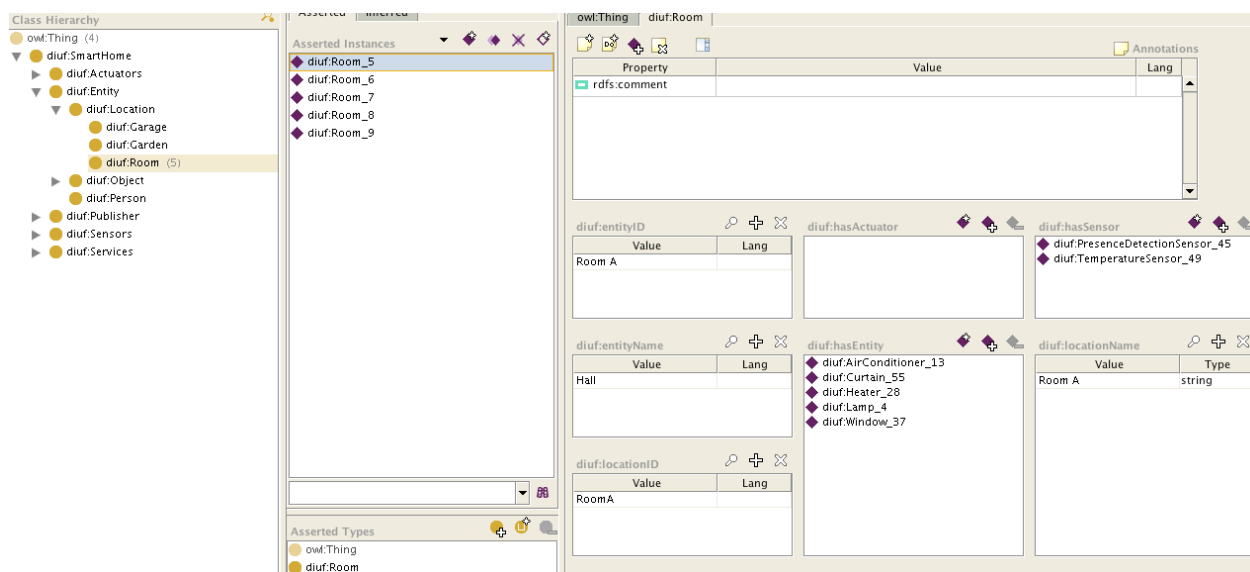


Figure 64 : Création d'une instance d'une pièce

Comme le scénario l'indique, cette pièce possède un radiateur (AirConditioner_13), des rideaux (Curtain_55), un chauffage (Heater_28), une lampe (Lamp_4) et une fenêtre (Window_57). Chaque instance a donc ses propres données et ses propres acteurs, senseurs, etc... L'instance AirConditioner_13 a donc plusieurs données comme un coolingmode, un fanmode qui indique le mode actuel utilisé par le radiateur, une entityId, un switchState qui permet d'indiquer si le radiateur est ouvert ou fermé, une temperature,... Il possède un actuateur de type AirConditionerController et deux senseurs : un de type ACSensor et un second de type SwitchSensor.

diuf:coolingMode		diuf:fanMode		diuf:hasActuator	
Value	Type	Value	Type		
AC	string	high	string	<div>◆ diuf:AirConditonerController_14</div>	

diuf:entityID		diuf:switchState		diuf:hasLocation	
Value	Lang	Value	Type		
AC13		Off	string	<div>◆ diuf:Room_5</div>	

diuf:entityName		diuf:temperature		diuf:hasSensor	
Value	Lang	Value	Type		
AC13		0.0	float	<div>◆ diuf:ACSensor_19</div> <div>◆ diuf:SwitchSensor_24</div>	

Figure 65 : Création d'une instance d'AirConditioner

Pour cette instance d'AirConditioner, son cooling mode a comme valeur « AC », son fanmode est à « high », son identifiant et son nom est « AC13 ». Il est actuellement éteint, car la propriété de donnée switchState est à « off », sa température est à zéro et il se situe dans la pièce nommée Room_5.

Un actuateur de type AirConditonerController possède quatre types d'actions:

1. adjustCoolingMode pour ajuster le mode de refroidissement
2. adjustTragetTemperature pour ajuster la température
3. switchOn pour allumer le radiateur
4. switchOff pour éteindre le radiateur

L'action switchOn_18 possède deux données essentielles : la location OWL-S et la location WSDL comme on le montre l'image ci-dessous. Ce sont donc les liens vers le fichier correspondant pour décrire le service web correspondant.

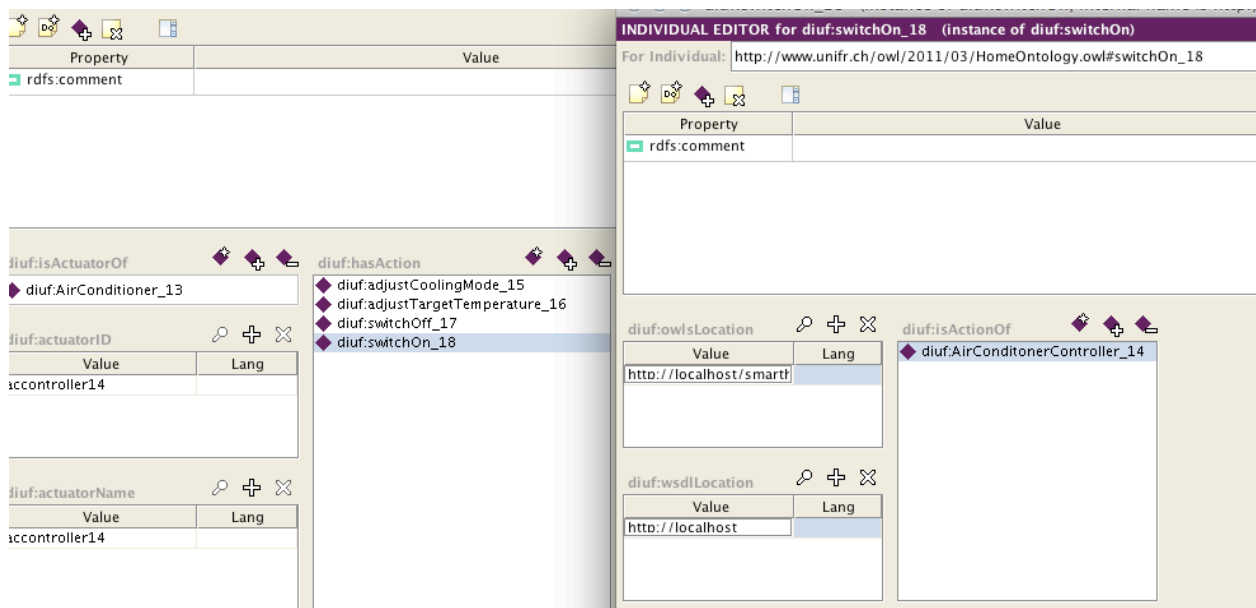


Figure 66 : WSDL et OWL-S locations pour l'action switchOf_18

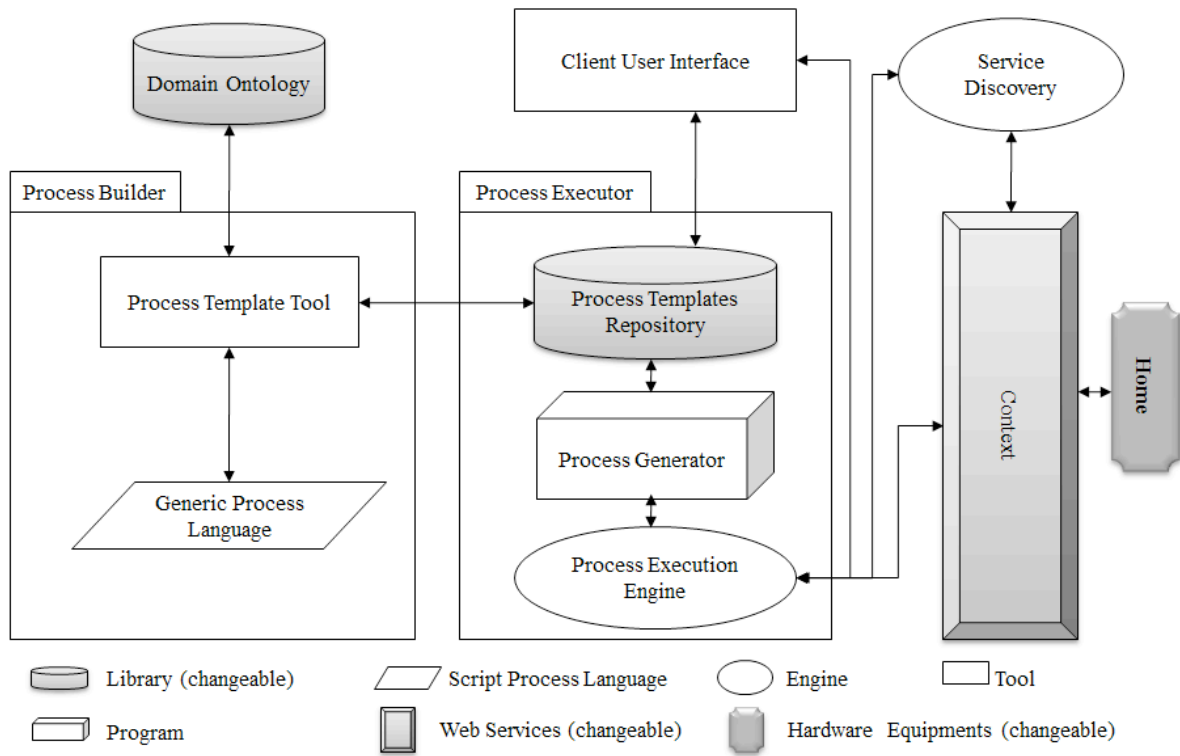
5.3 Le framework

5.3.1 L'architecture du framework

Le groupe Génie Logiciel de l'Université de Fribourg propose une solution sous la forme d'un framework afin de découvrir et utiliser de manière efficace les services fournis par l'environnement intelligent créé avec une ontologie [15]. La figure 67 montre une vue d'ensemble de ce framework qui se compose de plusieurs éléments et de fonctionnalités :

1. Process Template Repository : il permet à l'utilisateur de choisir un template de processus générique recommandé. Les templates sont définis en utilisant des ontologies de domaine, ainsi que le langage spécifique de domaine du groupe, qui sera présenté plus tard.
2. Process Generator : cette partie convertit le template choisi auparavant en un processus exécutable.
3. Au stade de l'exécution, les services concrets de l'environnement intelligent doivent être découverts selon leur ontologie.

4. Client User Interface : Cette interface offre la possibilité à l'utilisateur final de préciser ses préférences.
5. Execution Engine : C'est l'environnement d'exécution.



5.3.2 Un scénario

Afin d'illustrer au mieux l'utilité de cette approche, l'explication à travers la mise en œuvre d'un scénario simple qui se déroule dans une maison intelligente est nécessaire (voir figure 68). La maison contient un ensemble d'entités intelligentes qui ont été créées avec Protégé (des pièces, des radiateurs, des fenêtres, des lames, etc...). Le système de contrôle de la maison surveille les entités intelligentes dans le but de réaliser certains besoins de l'utilisateur comme l'économie d'énergie, l'amélioration de son confort, la sécurité, etc... Le scénario du groupe est simple et consiste à éteindre toutes les lampes une fois que le système est informé que les occupants de la maison sont à l'extérieur ou plutôt que plus personne ne se trouve dans la maison.

Il pourrait sembler que le scénario proposé demeure un peu trop simpliste pour mettre en avant tous les avantages de cette approche. Cependant, cette partie sert juste à décrire le concept de l'ontologie et l'ensemble du processus qui l'utilise afin d'instancier les entités intelligentes, qui peuvent ensuite être découvertes et gérées au sein d'un scénario écrit en GLP4SE. L'avantage de ce langage est qu'il est capable de gérer des scénarios beaucoup plus complexes avec des événements qui s'exécutent en parallèle et permet aussi de lister les entités intelligentes qui ont été filtrées avec des propriétés comme toutes les fenêtres situées dans une certaine pièce ou le radiateur associé à la chambre X. Pour un aperçu plus détaillé, il est possible de trouver une version plus complexe du scénario à [11] dans lequel les lampes s'éteignent en fonction de la présence de quelqu'un ou non dans les différentes pièces, il est possible de maintenir la température dans une fourchette raisonnable ou encore de contrôler l'eau afin que l'on ne l'utilise pas plus qu'un intervalle de temps donné.

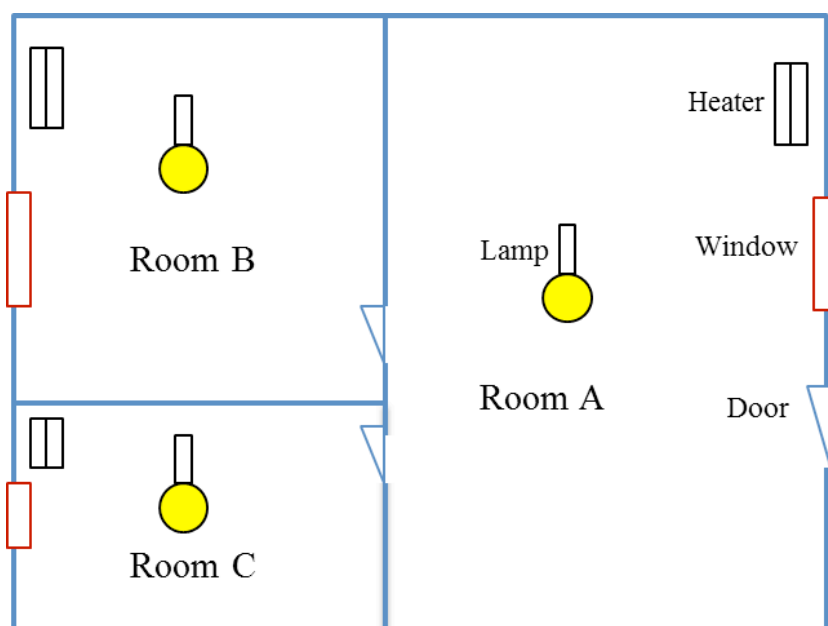


Figure 68: La maison intelligente du scénario présenté (pris dans [15])

Afin d'effectuer ce scénario basique, le système doit en premier connaître toutes les lampes qui sont dans la maison, mais aussi connaître les acteurs et senseurs de chaque lampe. Le senseur est utilisé pour savoir si la lampe est allumée ou éteinte et l'acteur permet d'éteindre la lampe au cas où celle-ci est allumée. Ce scénario est effectué en trois étapes comme le montre la figure 69:

1. l'instanciation de l'ontologie
2. les actuateurs et les senseurs sont accessibles via les fichiers WSDL et OWL-S qui sont publiés dans un registre.
3. le moteur d'exécution du framework utilise le moteur de découverte (discovery engine) qui permet de découvrir l'ontologie dans le but de trouver tous les services nécessaires et leurs entités pour le scénario. Des requêtes SPARQL sont utilisées à ce niveau.

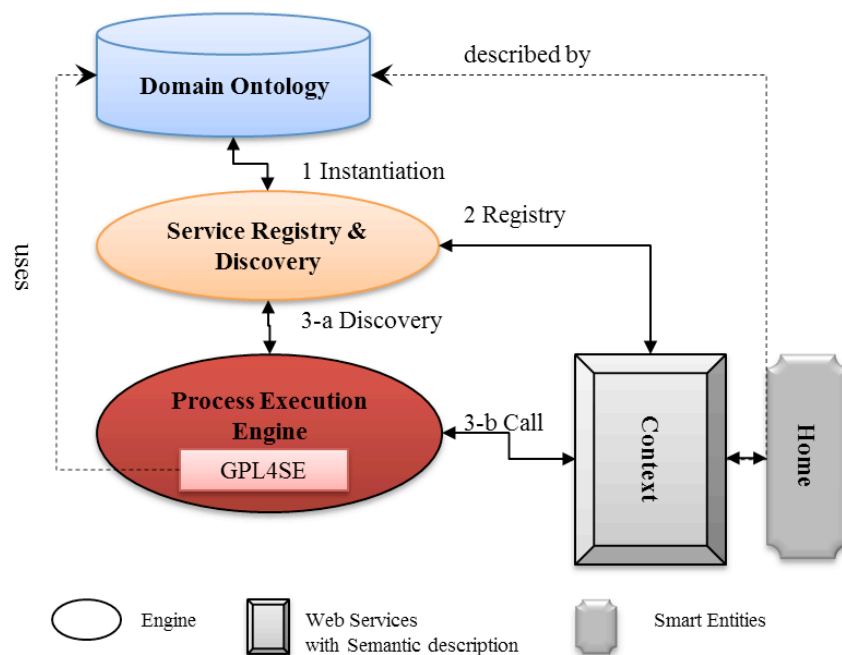


Figure 69: Registre des services, Discovery et tous les processus appelés (pris dans [15])

5.3.3 L'instanciation de l'ontologie

L'instanciation de l'ontologie et donc de la maison intelligente a été expliquée dans le chapitre 5.2.5. Trois pièces ont été créées et chacune est composée de plusieurs objets. Pour cette maison intelligente, la configuration de l'environnement correspond à l'instanciation des classes de l'ontologie présentées sur la figure 71. La hiérarchie des classes et le fait que les entités soient intelligentes parce qu'elles sont attachées à des actuateurs ou/et à des senseurs ont déjà été expliqués dans les chapitres précédents. La figure 70 représente un extrait du fichier OWL qui contient l'instanciation de la Room A.

- Lignes 1 à 32 définissent que l'instance *Room_1* qui a un entityID *Room A*, est de type de la classe *Room*.
- Lignes 3 à 20 définissent que la pièce *Room A* possède l'instance *Lamp_1*.
- Lignes 5 à 18 définissent que l'instance *Lamp_1* possède un actuateur nommé *LampController_1*.
- Lignes 7 à 10 définissent que l'instance *LampController_1* possède l'action *switchOff*.
- Lignes 21 à 31 définissent que l'instance *Room_1* possède le senseur *AccessSensor_1* avec un « event publisher ».

```

1. <diuf:Room rdf:ID="Room_1">
2.   <diuf:entityID rdf:datatype="#string">Room A</diuf:entityID>
3.   <diuf:hasEntity>
4.     <diuf:Lamp rdf:ID="Lamp_1">
5.       <diuf:hasActuator>
6.         <diuf:LampController rdf:ID="LampController_1">
7.           <diuf:hasAction>
8.             <diuf:switchOff rdf:ID="switchOff_1">
9.               <diuf:owlsLocation rdf:datatype="#string"></diuf:owlsLocation>
10.            </diuf:switchOff>
11.            . . . . .
15.          </diuf:hasAction>
16.          <diuf:actuatorID rdf:datatype="#string">4754</diuf:actuatorID>
17.        </diuf:LampController>
18.      </diuf:hasActuator>
19.    </diuf:Lamp>
20.  </diuf:hasEntity>
21.  <diuf:hasSensor>
22.    <diuf:AccessSensor rdf:ID="AccessSensor_1">
23.      <diuf:hasPublisher>
24.        <diuf:AccessEventPublisher rdf:ID="AccessPublisher_1">
25.          <diuf:hasEvent>
26.            <diuf:AccessEvent rdf:ID="AccessEvent_1"></diuf:AccessEvent>
27.          </diuf:hasEvent>
28.        </diuf:AccessEventPublisher>
29.      </diuf:hasPublisher>
30.    </diuf:AccessSensor>
31.  </diuf:hasSensor>
32. </diuf:Room>

```

Figure 70: Code extrait d'une instanciation d'une lampe intelligente et de ses services (pris dans [15])

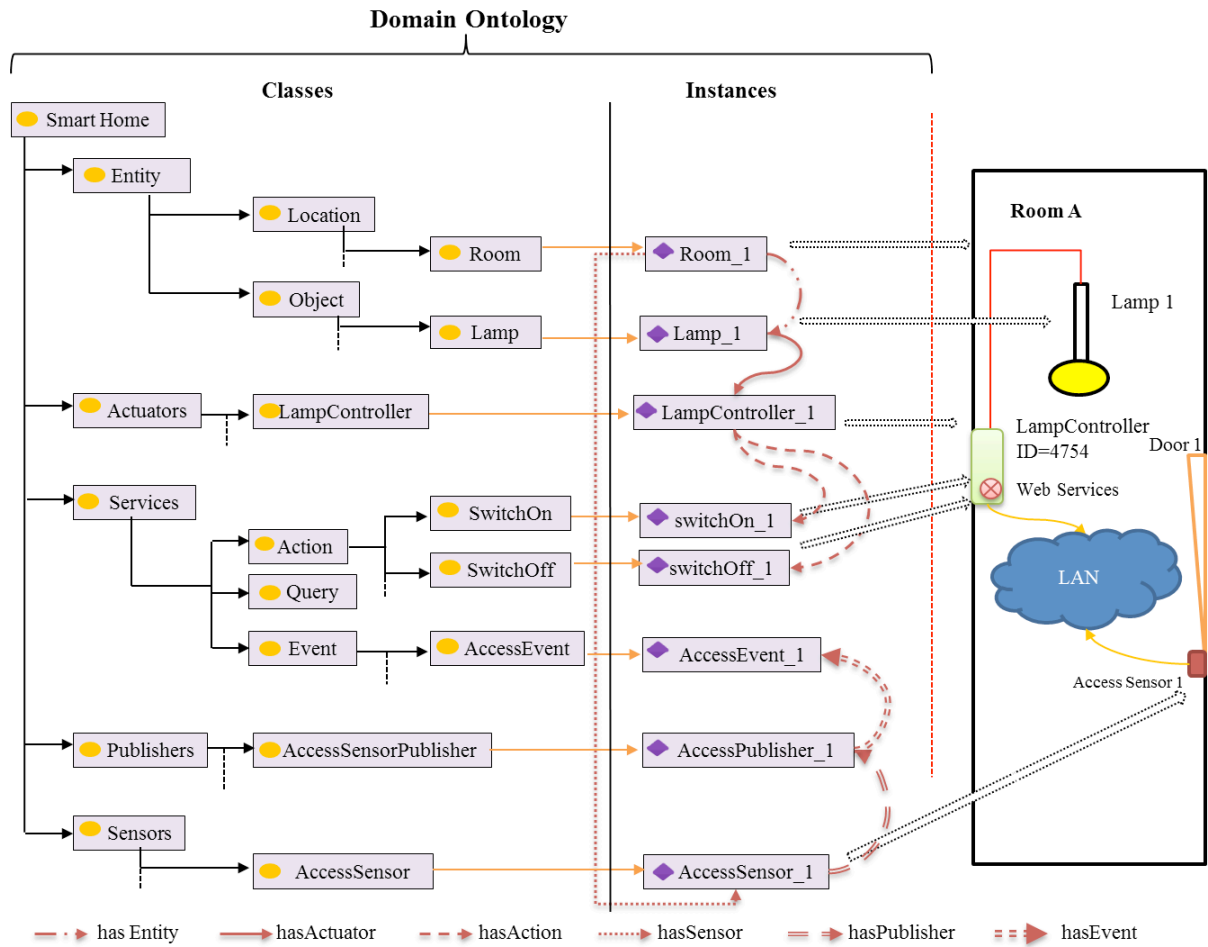


Figure 71: L'ontologie de domaine simplifiée et son instanciation pour un maison spécifique (pris dans [15])

5.3.4 Le registre des services

Une fois que l'ontologie a été instanciée, c'est un actuateur ou un senseur associé à une entité qui commence le processus d'enregistrement. L'actuateur ou le senseur utilise un protocole SOAP afin de communiquer avec le registre. Il fournit au registre différentes informations comme son identifiant, son nom, son emplacement, ses services et leurs adresses WSDL et OWL-S. Tous les actuateurs et les senseurs s'inscrivent dans ce registre. Une fois que cela est fait, les services Web sont désormais liés à ce registre.

Un extrait d'un message SOAP envoyé par un actuateur de type LampController est donné dans la figure 72. Il montre que l'actuateur est identifié par son ID (4754) et qu'il fournit deux

actions: switchOn et switchOff. Ces deux services peuvent être appelés en utilisant leur adresse WSDL et OWL-S.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body xmlns:ns1="http://ws/"><ns1:registreActuator><id>4754</id>
    <service><name>switchOn</name>
      <owlURL>http://diufpc10:8080/SmartHome/LivingRoomLamp.owl</owlURL>
      <wsdlURL>http://diufpc10:8080/SmartHome/LivingRoomLamp?WSDL</wsdlURL>
    </service>
    <service><name>switchOff</name>
      <owlsURL>http://diufpc10:8080/SmartHome/LivingRoomLamp.owl</owlsURL>
      <wsdlURL>http://diufpc10:8080/SmartHome/LivingRoomLamp?WSDL</wsdlURL>
    </service>
  </ns1:registreActuator></soap:Body>
</soap:Envelope>
```

Figure 72: Message SOAP envoyé par un actuateur LampController (pris dans [15])

Durant cette phase d'insertion où les actuateurs et les senseurs s'enregistrent, des requêtes SPARQL sur l'ontologie sont utilisées par le framework afin de trouver les actuateurs et les senseurs concernés.

Le moteur de registre va d'abord créer son registre d'après les classes définies dans l'ontologie. La première requête va permettre de sélectionner une classe d'entités et le ou les actuateurs correspondants et une seconde pour le ou les senseurs correspondants. Le problème est que dans l'ontologie, certaines classes d'objets permettent d'avoir plusieurs actuateurs et plusieurs senseurs et d'autres qu'un seul. Une des solutions est de séparer ces différents cas et de créer une requête pour chaque cas.

Dans le cas où la classe permet de posséder qu'un seul actuateur, la requête SPARQL est la suivante:

```

1      SELECT ?entity ?actuator WHERE {
1      ?entity a owl:Class ;
2      rdfs:subClassOf ?restriction.
3      ?restriction owl:onProperty diuf:hasActuator ;
4      owl:allValuesFrom ?actuator.
5      }

```

On parle ici d'une classe et non d'instances. Il faut donc que la variable entity soit une classe. La propriété raccourcie "a" a été utilisée à la place de rdf:type et owl:Class pour définir ceci. La propriété owl:onProperty spécifie que je veux une propriété d'objet hasActuator et owl:allValuesFrom permet de récupérer les restrictions universelle liées à cette classe et cette propriété.

Pour le cas où la classe permet de lier plusieurs actuateurs, la requête est un peu modifiée. Il faut retourner les actuateurs sous la forme d'une liste grâce à la propriété owl:unionOf.

```

1      SELECT ?entity ?list WHERE
2      ?entity a owl:Class ;
3      rdfs:subClassOf ?restriction.
4      ?restriction owl:onProperty diuf:hasActuator ;
5      owl:allValuesFrom ?actuator.
6      ?actuator owl:unionOf ?list.
7      }

```

Ces deux requêtes SPARQL sont presque identiques pour les actions, les senseurs et les requêtes. Il suffit juste d'indiquer quelle propriété on veut pour la restriction en modifiant la ligne 4. Le triplet ?restriction owl:onProperty diuf:hasSensor sera utilisé pour connaître les senseurs, et le triplet ?restriction owl:onProperty diuf:hasAction pour les actions, etc...

Ensuite, le moteur de registre va enregistrer les instances de ces classes afin de connaître toutes les instances d'actuateurs et de senseurs mais ceux-ci ne communiquent pas leurs adresses WSDL et OWL-S pour le moment. Quelques requêtes sont nécessaires pour effectuer cela. Il faut enregistrer chaque instance d'actuateurs ainsi que leur(s) actions correspondantes et chaque instance de senseurs ainsi que leur(s) requêtes.

Une fois que tous les actuateurs et tous les senseurs sont insérés dans le registre, ils vont communiquer leur adresse WSDL et leur adresse OWL-S également grâce à des requêtes SPARQL qui sont plus au moins identiques aux requêtes utilisées pour visualiser l'environnement et qui sont expliquées ci-dessous.

Visualisation de l'environnement

Le framework propose de visualiser en temps réel les objets de la maison intelligente avec leur(s) actuateur(s) et senseur(s) grâce à une interface graphique. Elle se compose sous la forme d'un arbre comme on peut le voir sur la figure 73. Afin de créer l'arbre, il faut parcourir toute l'ontologie et récupérer quelques informations comme les classes des entités, des lieux, des actuateurs, des senseurs, des actions, ... et leurs instances. Deux requêtes sont utilisées à ce niveau. La première pour sélectionner toutes les entités intelligentes et leur(s) actuateur(s), la seconde les entités et leur(s) senseur(s).

```
1      SELECT distinct ?locationType ?location ?entity ?entityName ?entityType
      ?actuator ?id ?action ?actionType ?wsdlLocation WHERE {
2      ?entity rdf:type ?entityType.
3      ?location diuf:hasEntity ?entity.
4      ?location rdf:type ?locationType.
5      ?entity diuf:entityName ?entityName.
6      ?entity diuf:hasActuator ?actuator.
7      ?actuator diuf:actuatorID ?id.
8      ?actuator diuf:hasAction ?action.
9      ?action a ?actionType.
10     ?actionType rdfs:subClassOf diuf:Action.
11     ?action diuf:wsdlLocation ?wsdlLocation
12     }
```

Cette première requête sélectionne toutes les instances intelligentes distinctes qui ont un ou plusieurs actuateurs ainsi que toutes leurs informations comme leur lieu, leurs actions, la location WSDL, ... La variable `wsdlLocation` contient l'adresse WSDL de chaque action.

```
1      SELECT  ?locationType ?location ?entityType ?entity ?entityName ?sensor
          ?id ?query ?queryType ?wsdlLocation WHERE {
2      ?entity rdf:type ?entityType.
3      ?location diuf:hasEntity ?entity.
4      ?location  rdf:type  ?locationType.
5      ?entity diuf:entityName ?entityName.
6      ?entity diuf:hasSensor ?sensor.
7      ?sensor diuf:sensorID ?id.
8      ?sensor diuf:hasQuery ?query.
9      ?query a ?queryType.
10     ?queryType rdfs:subClassOf diuf:Query.
11     ?query diuf:wsdlLocation ?wsdlLocation
12     }
```

Cette seconde requête fait la même chose mais cette fois-ci elle recense les instances qui ont des senseurs. Avec ces deux requêtes, on peut stocker toutes ces informations dans une liste. La maison peut être hiérarchisée par pièce. Dans chaque pièce demeure les instances d'objets qui ont eux des acteurs et des senseurs qui possède leur(s) propre(s) action(s) ou requête(s). Il suffit ensuite de lire cette liste pour afficher l'ontologie sous la forme d'un arbre.

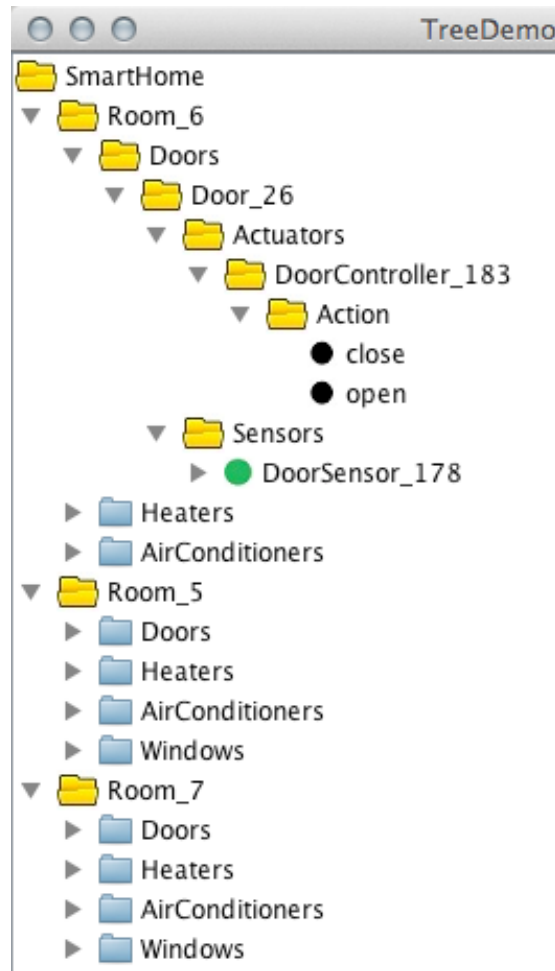


Figure 73: Représentation de l'instanciation de la maison intelligente sous la forme d'un arbre

5.3.5 Le processus de découverte : l'utilisation des objets intelligents

Une fois que l'ontologie est instanciée et que tous les acteurs et senseurs sont dans le registre, on peut créer et lancer un scénario en utilisant le langage GPL4SE. Il sera alors possible d'interagir avec les entités intelligentes. Le Discovery Engine est alors utilisé pour questionner le registre. Il va automatiquement traduire la requête GPL4SE en SPARQL et ensuite effectuer ce qui est demandé. Avec GPL4SE, il est possible de découvrir et d'interagir avec différentes entités intelligentes, de créer des boucles, de déclarer des variables, de copier et assigner des valeurs ainsi qu'enregistrer et attendre des événements. Les scénarios peuvent être exécutés séquentiellement ou en parallèle. Pour illustrer ce processus de découverte et l'interaction entre GPL4SE et les entités intelligentes, un extrait de code GPL4SE et sa requête correspondante en SPARQL est donnée dans la figure 74. La structure du programme est divisé en trois sections: une section pour la découverte (discovery), une deuxième pour le

registre d'événement et une troisième pour l'exécution. La première section est le processus de découverte sémantique qui commence par l'envoi d'une requête à l'aide du mot-clé « discover ». La requête a deux paramètres: la description sémantique des entités requises et leur emplacement (facultatif). Une fois que la requête est reçue par le moteur de découverte, elle est automatiquement traduite par le moteur de découverte dans une requête SPARQL pour trouver toutes les entités intelligentes dans la pièce A. La deuxième section a pour but de souscrire le processus à un AccessEvent afin d'être informé de tout nouveau changement qui pourrait se produire. Quand un événement d'accès est déclenché, le processus exécute alors l'action onEvent appropriée.

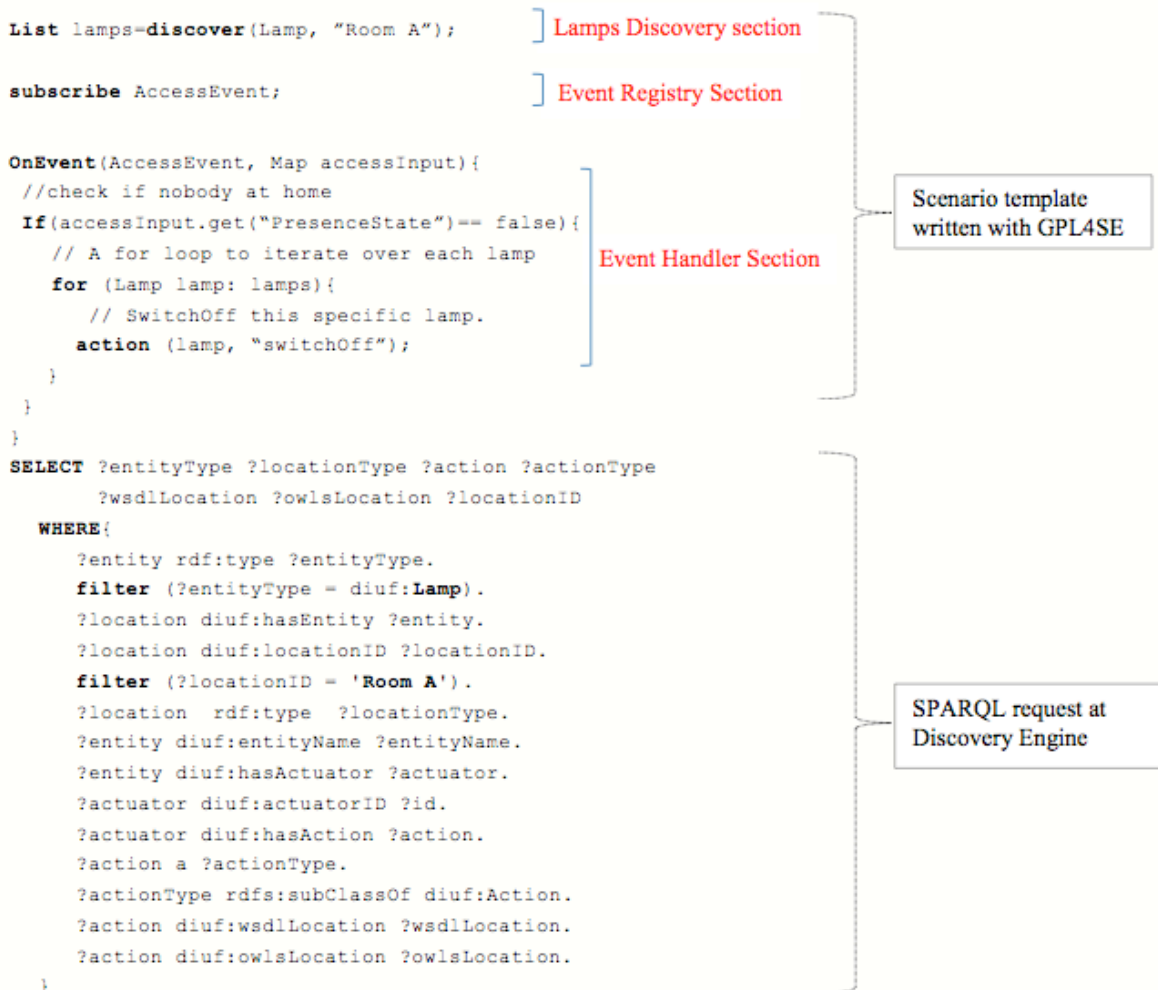


Figure 74: Extrait de code GPL4SE et la requête SPARQL correspondante (pris dans [15])

On remarque que la requête SPARQL traduite a pour but de connaître toutes les instances de Lamp qui sont dans la pièce nommé « Room A » ainsi que leurs services et les liens vers les fichiers WSDL et OWL-S pour connaître le service web correspondant. Cela correspond parfaitement à la requête GPL4SE utilisée. En effet, il faut d’abord découvrir toutes les lampes d’une pièce de la maison. Ensuite on questionne le senseur “PresenceDetectionSensor” via sa requête getPresenceState pour savoir si quelqu’un se trouve dans la pièce ou non. Si ce n’est pas le cas, on éteint toutes les lampes de la pièce grâce à l’action switchOff de l’actuateur correspondant.

Deux filtres sont utilisés dans la requête SPARQL : le premier pour ne retourner que les instances qui appartiennent à la classe Lamp et un second pour qu’elles soient dans la pièce « Room A ». Les autres triplets utilisent des propriétés RDFS, RDF ou de l’ontologie. Il faut d’abord trouver les entités d’un certain type, qui se trouvent dans un certains lieu, puis sélectionner l’actuateur ou les actuateurs et les actions. Finalement, il faut sélectionner l’url des fichiers WSDL et OWL-S des services.

Pour le scénario de la maison intelligente, les actuateurs et les senseurs sont simulés et décrits dans les Web Service Interfaces. Chaque actuateur et senseur possèdent une interface qui fournit ses services (requêtes, événements et actions). Une interface graphique permet de visualiser les différents événements appelés et les services invoqués. La figure 75 montre une capture d’écran de cette interface. La partie du haut montre la réaction du système selon les action de scénario créé ou choisi. Les différents résultats de scénarios comme le nom des instances concernées ainsi que l’action demandée par exemple s’affichent. La partie du bas propose différents services comme par exemple de changer la température d’une pièce ou de fermer le robinet d’une douche et de publier artificiellement ces événements.

Smart Home

N° : 5

14:36:13

Heater

Location : Room A (Living Room)

Change State To: Off

Access Control System	Presence Event	Calendar Events	Temperature Changing Event	Shower Event	Lock Event	Motion Event
<div>LockState</div> <div>unl...</div> <div>Location</div> <div>Entr...</div> <div>AccessCode</div> <div>3221</div> <div>StayMode</div> <div>abs...</div> <div>DoorState</div> <div>close</div> <div>Publish</div>	<div>PresenceState</div> <div>noB...</div> <div>Location</div> <div>A</div> <div>Publish</div>	<div>DayTime</div> <div>night</div> <div>Publish</div>	<div>Temperature</div> <div>-40</div> <div>Location</div> <div>A</div> <div>Publish</div>	<div>ValveState</div> <div>unl...</div> <div>Location</div> <div>C</div> <div>Publish</div>	<div>LockState</div> <div>unl...</div> <div>Location</div> <div>A</div> <div>Publish</div>	<div>MotionState</div> <div>noB...</div> <div>Location</div> <div>A</div> <div>Publish</div>

Figure 75: L'interface de la maison intelligente pour controler les invocations de services

6

Conclusion

L'utilisation d'une ontologie structurée et des Web Services est plutôt une idée séduisante. Les langages du Web sémantique sont de plus en plus répandus du fait que le Web sémantique est mené par le W3C et qu'il répond parfaitement à la demande actuelle du partage et de la réutilisabilité des données Web entre plusieurs applications. Des bibliothèques pour OWL et Protégé par exemple existent dans les principaux langages comme java, C, .Net,...

Du point de vue de la programmation, tous les langages du Web sémantique sont faits pour être utilisés ensemble et c'est un gros avantage. En effet, on peut facilement combiner du RDF, du RDFS et du OWL. Protégé est l'outil le plus utilisé pour construire une ontologie et il propose une interface ergonomique et facile à comprendre. Le point faible réside dans la lisibilité et la compréhension du RDF, de OWL et de SPARQL. En effet, ces langages ne sont pas très compliqués et on apprend assez vite les bases, mais quand on regarde le code, ça reste assez incompréhensible du premier coup d'œil. Il existe néanmoins de nombreux tutoriels et exemples pour se familiariser avec ce type de langage.

Durant ce travail, après m'être familiarisé avec les langages du Web sémantique et avec les ontologies, j'ai contribué à créer une ontologie de domaine d'une maison intelligente qui puisse d'abord répondre aux exigences du scénario du groupe Génie Logiciel, puis de l'étendre. Cette ontologie fait partie de la base du framework et est essentielle. Puis j'ai créé plusieurs requêtes SPARQL pour interroger l'ontologie afin qu'elles soient utilisées dans différentes parties du framework comme dans la phase d'insertion des instances, dans la phase de visualisation de l'environnement et enfin dans le moteur de découverte.

Personnellement, j'ai trouvé très intéressant d'apprendre et de travailler avec ces langages qui m'étaient inconnus auparavant. Je pense que ces langages seront de plus en plus utilisés dans certains domaines à l'avenir. Le Web sémantique offre une grande flexibilité et améliore la qualité d'échange de connaissances ce qui est très prisé de nos jours. Par contre, le Web sémantique peut paraître à première vue complexe et rigide. On peut en effet avoir du mal à en trouver l'utilité avant de l'avoir utilisé.

Références

- [1] Carmen Martinez-Cruz, Ignacio J. Blanco, M. Amparo Vila. *Ontologies versus relational databases: are they so different? A comparison*. *Artificial Intelligence Review* 38, Netherlands, 2012.
- [2] Web sémantique – Wikipedia. http://fr.wikipedia.org/wiki/Web_sémantique (dernière consultation le 06.05.2014)
- [3] OWL Web Ontology Language – W3C. <http://www.w3.org/TR/owl-features/> (dernière consultation le 10.06.2014)
- [4] RDF – Wikipedia. http://fr.wikipedia.org/wiki/Resource_Description_Framework (dernière consultation le 22.04.2014)
- [5] Fabian Gandon, Catherine Faron-Zucker et Olovier Corby. *Le web sémantique Comment lier les données et les schémas sur le web?* Dunod. Paris: Dunod, 2012.
- [6] Horridge, Matthew. *A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools*. Université de Manchester, 2011.
- [7] Project Ara. <http://www.projectara.com> (dernière consultation le 03.04.2014)
- [8] SPARQL Query Language for RDF – W3C. <http://www.w3.org/TR/rdf-sparql-query/> (dernière consultation le 20.06.2014)
- [9] Modifiers for SPARQL – W3C. <http://www.w3.org/TR/rdf-sparql-query/#solutionModifiers>
- [10] Opérateurs de SPARQL – W3C. <http://www.w3.org/TR/rdf-sparql-query/#SparqlOps> (dernière consultation le 20.06.2014)
- [11] Abdaladhem Albreshne, Ayoub Ait Lahcen and Jacques Pasquier. *A Framework and its Associated Process-Oriented Domain Specific Language for Managing Smart Residential Environments*, In *International Journal of Smart Home*, Vol. 7, No. 6, pp. 377-392, Fribourg, 2013
- [12] OWL-S – Wikipedia. <http://en.wikipedia.org/wiki/OWL-S> (dernière consultation le 05.06.2014)
- [13] WSDL – Wikipedia http://en.wikipedia.org/wiki/Web_Services_Description_Language (dernière consultation le 05.06.2014)

- [14] Consultation d'ontologies OWL-DL au travers de SPARQL - Laboratoire d'informatique, de robotique et de microélectronique de Montpellier (LIRMM) <http://www.lirmm.fr/~mougenot/Enseignement/FMIN321/PartieSparql.pdf> (dernière consultation le 19.06.2014)
- [15] Abdaladhem Albreshne, Ayoub Ait Lahcen, Jacques Pasquier. *Using a Residential Environment Domain Ontology for Discovering and Integrating Smart Objects in Complex Scenarios*, Fribourg, 2014.
- [16] Abdaladhem Albreshne, Jacques Pasquier. *Semantic-Based Semi-Automatic Web Service Composition Proceedings of the Fifth Libyan Arab International Conference On Electrical and Electronic Engineering LAICEEE*, Vol. 1, pp. 603-615, Tripoli, Libya, 2010.
- [17] Abdaladhem Albreshne, Jacques Pasquier. *A Template-Based Semi-Automatic Web Services Composition Framework: Case Study of Smart Home Management*, Proceedings of the PhD Symposium at the 9th IEEE European Conference on Web Services ECOWS11, Technical Report 2011/08, pp. 11-17, Lugano, Suisse, 2014.