TER - Vers une plus grande flexibilité dans le dialogue avec un habitat intelligent

Clément Didier, Sybille Caffiau, Alexandre Demeure,
François Portet et Nicolas Bonnefond

13 Juin 2017

Résumé

Nous nous interessons dans ce document à la compréhension des expressions en langage naturel formulées par les habitants d'un foyer, pour générer des règles d'automatisation de leur habitat, sans expertises techniques nécessaires.

Introduction

Les avancées technologiques de ces dernières décénnies permettent d'imaginer l'intégration des nouvelles technologies dans la plus part des activités humaines.
L'habitat intelligent est le support technologique proposé pour les activités domestiques. Il peut être défini par un "ensemble de services de l'habitat assurés par des systèmes réalisant plusieurs fonctions, pouvant être connectés entre eux et à des réseaux internes et externes de communication" [1]. En effet, l'habitat intelligent correspond à un ensemble d'objets connectés qui peuvent être employés conjointement pour exécuter des actions automatisées. D'après [2] l'habitat intelligent peut également être défini comme "l'ensemble des services offerts aux occupants d'un logement fondés sur l'échange d'informations et permettant d'accéder à un nouvel art de vivre". Ainsi il n'est plus seulement question de technologie, mais aussi des besoins et du confort des habitants. Pour les prendre en compte, une approche est de leur laisser la possibilité de programmer leur propre habitat. L'approche du End-User Development [3] permet aux habitants de personnaliser les automatismes de leur habitat et ainsi de lui donner un comportement qui répond à leurs besoins. Ces automatismes sont programmés par un ensemble de règles peu intuitives pour des personnes sans expertises en informatique (*ref.* Figure 1).
Pour répondre à cette problématique, une solution est de permettre aux habitants de construire ces règles par l'usage d'un langage pseudo-naturel [4], c'est-à-dire un langage de programmation de haut niveau. Cette solution permet de s'abstraire du vocabulaire de la programmation, mais pas des mécanismes algorithmiques (structuration de programmes, connecteurs logiques...).
L'approche dans laquelle nous nous intégrons a pour objectif de laisser l'habitant s'exprimer avec ses propres mots et structures de phrases, puis de l'accompagner par un mécanisme de dialogue jusqu'à ce que le système soit capable de générer un programme compilable. Une première solution a été proposée pour suivre cette approche [5]. Le dialogue mis en place permet de désambiguïser la requête formulée, pour des règles très limitées. En effet, la solution ne prend pas en compte les instances multiples d'objets dans l'habitat, ni les conditions composées. De

plus, la solution implémentée ne dispose pas d'une structure adaptée à l'évolution du système et à l'ajout d'extensions. Afin d'obtenir un dialogue plus flexible, nous proposons une réalisation basée sur un langage "pivot" permettant d'abstraire la complexité de communication entre la machine et le système de gestion de l'habitat. Nous proposons également une phase d'analyse du texte en langage naturel différente avec l'usage d'outils récents gratuits, Open Source [1] et pouvant être installés localement.
La première section présente le contexte lié au traitement automatique du langage naturel pour programmer l'habitat. L'approche suivie dans [5] et ses limitations seront également présentées dans cette première section. La seconde section est consacrée à la présentation de la nouvelle chaîne de traitement proposée. Celle-ci s'appuie sur le modèle "pivot" exprimant les connaissances technologiques et la sémantique associée pour les habitants. Ce modèle sera présenté dans la section 3, avant de conclure.

1 Le traitement automatique du langage naturel pour la programmation de l'habitat

1.1 Le langage naturel

Le langage naturel correspond au langage employé pour communiquer par les Hommes. Celui-ci est défini par un ensemble de règles grammaticales et un très grand vocabulaire en constante évolution. Il est également très complexe puisqu'il permet d'exprimer une même notion avec un grand nombre de formulations différentes, ce qui peut amener à des problèmes d'interprétation (ambiguités du langage). En prenant par exemple la requête "éteins la lumière", celle-ci peut exprimer l'extinction d'une seule lampe, mais aussi l'extinction de toutes les lampes. Le contexte dans lequel la phrase est utilisée peut permettre la désambiguisation. En effet, cette expression peut dépendre d'un contexte particulier, auquel cas, l'interlocuteur ou le lecteur doit en être informé pour en tenir compte dans son interpretation. En reprenant l'exemple des lumières, il faut éventuellement connaître la pièce où les lumières doivent être éteintes, mais également avoir connaissance de tous les éléments pouvant faire de l'éclairage et de comment les éteindres (informations sur la procédure etc.). Ainsi le langage naturel est un langage ambiguë et complexe que les machines ne peuvent directement interprêter (contrairement à des langages formels, tels que des langages informatiques). Il est nécessaire pour qu'il soit compris par la machine que l'Homme explicite la sémantique du vocabulaire et apporte les informations contextuelles.

1.2 Les règles OpenHab

Une règle OpenHab (*ref.* Figure 1) est un ensemble d'instructions permettant de réaliser des automatismes pour l'habitat (par exemple, allumée la télévision lorsqu'il est midi). Une telle règle se définit par un entête spécifiant son nom et de deux blocs d'instructions (appelés respectivement "bloc conditionnel" et "bloc d'éxecution"). Le premier bloc determine la condition d'éxecution de la règle (par exemple "Time is noon" dans le cas où elle doit se lancer à midi). Tandis que le second caractérise l'ensemble des instructions qu'elle doit éxecuter si la condition est validée (par exemple "sendCommand(tv_power, ON)" dans le cas où il faut allumer la télévision).

1. Logiciel autorisant et permettant le libre accès à son code source technique

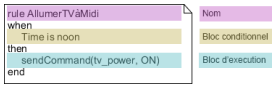


FIGURE 1 – Exemple de règle simple OpenHab

1.3 Le traitement automatique du langage naturel

L'objectif de nos travaux est de transcrire une requête formulée en langage naturel, en une requête en langage formel compilable par le logiciel d'automatisation de l'habitat (OpenHab [2]) en levant les ambiguïtés du langage. Une première solution [5] ayant suivie cette approche reprend les principes de l'interaction en optant pour l'emploi d'un "feedback" [3] sur l'expression formulée par l'habitant. Il permet de renseigner l'habitant sur la validité de sa requête, mais également de lui demander des précisions si une désambiguïsation est nécessaire. Afin de comprendre l'expression formulée par l'habitant, cette solution utilise les trois niveaux d'analyse du traitement automatique du langage naturel. Ces méthodes se décomposent généralement en trois phases séquentielles : l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.

L'analyse lexicale est une étape de traitement d'une expression constituée de mots et consiste à trouver les propriétés lexicales de chacun de ces mots, telles que leur catégorie (nom, verbe, etc.) et leur lemme (forme parente des mots, par exemple le lemme de "allume" est "allumer"), par l'usage d'un grand ensemble de données lexicales. La recherche de correspondance dans l'ensemble de données définit la limite principale de cette phase, puisqu'il s'agit d'une recherche coûteuse et dépendante de la taille de l'ensemble et des données qui le compose (dictionnaire de plusieurs centaines de miliers de mots avec leurs différentes variations possibles). Il est également nécessaire de mettre à jour cet ensemble lorsque la langue évolue, par l'apparition ou la modification de mots du langage par exemple.

La solution [5] dispose d'une chaîne de traitement commençant par cette étape d'analyse, en utilisant l'outil TreeTagger [4]. Dans nos travaux, nous ne modifierons pas la phase d'analyse lexicale.

L'analyse syntaxique quant à elle est une phase d'étiquetage, qui associe une catégorie syntaxique pour chaque élément appelé "token", résultant de l'analyse lexicale. Cette phase d'analyse peut conduire au résultat suivant au format CONLL [5] (*ref.* Figure 2).

ID	FORM	LEMA	UPOSTAG	XPOSTAG	FEATS	HEAD	DEPREL	DEPS	MISC
1	à	_	ADP	ADP	_	2	case	_	_
2	midi	_	NOUN	NOUN	_	3	nmod	_	_
3	allumer	_	VERB	VERB	_	0	ROOT	_	_
4	le	_	DET	DET	_	5	det	_	_
5	tv	_	NUM	NUM	_	3	nsubj	_	_

FIGURE 2 – Résultat d'analyse syntaxique avec l'outil SyntaxNet (format CONLL)

Ce format résultat dispose d'autant de lignes que de "tokens" analysés et de dix colonnes donnant toutes un type d'information différent sur chacun de ces "tokens" :

2. Logiciel de contrôle et d'automatisation d'habitat domotisé, *https ://www.openhab.org/*
3. Représente un retour d'information ou d'expérience auprès d'un utilisateur
4. http ://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/
5. http ://universaldependencies.org/format.html

1. **ID** L'index unique croissant du token, en respectant l'ordre de lecture dans l'expression analysée.
2. **FORM** La valeur du mot dans l'expression, sans aucun traitement.
3. **LEMMA** Le lemme du mot (ici, la colonne est vide car l'outil SyntaxNet ne permet pas de les obtenirs).
4. **UPOSTAG** [6] Le mot clé de la propriété lexicale et grammaticale du mot.
5. **XPOSTTAG** Quasiment identique à la colonne précédante, mais en prenant en compte les spécificités du langage.
6. **FEATS** Liste les caractéristiques morphologiques du mot. (SyntaxNet ne donne pas cette information)
7. **HEAD** L'index du token auquel le mot dépend.
8. **DEPREL** [7] La relation syntaxique universelle du mot.
9. **DEPS** [8] Une seconde forme de relations grammaticales.
10. **MISC** Les informations et annotations supplémentaires.

L'analyse syntaxique permet de désambiguïser une partie des ambiguïtés résultantes de l'analyse lexicale, grace a l'information disponible à la 4ème colonne de la figure ci-dessus (*ref.* Figure 2). Cette information donne la propriété de chaque "token" et de son utilité dans la phrase. Le format permet également de donner les relations de dépendance de chacun d'eux (7ème colonne), dont une représentation est donnée ci-dessous (*ref.* Figure 3).

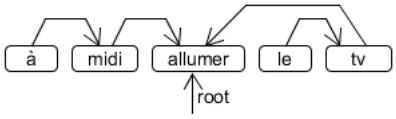


FIGURE 3 – Représentation des dépendances entre "tokens"

Cette phase nécessite un apprentissage avant d'être employée, ce qui implique l'usage d'une grande quantité de données annotés. Le résultat dépend des domaines d'application de ces données, mais également de leur quantité, qui influe à la fois sur la qualité des résultats, puisqu'il s'agit très souvent d'approches stochastiques [9], mais aussi sur la vitesse de traitement.
Pour permettre l'adaptation aux différents foyers (et donc aux habitants) sans disposer de quantité de données, la solution [5] propose de passer par une phase de pré-traitement nommée "normalisation" qui lui permet de simplifier les éléments (terminologies, négations, expressions impersonnelles etc.) et de remplacer les mots dont les synonymes sont présents dans un dictionnaire (adaptation au foyer), par des variables intégrées au système (par exemple "ventilo" sera remplacé par "ventilateur").
L'analyse sémantique est la troisième étape du traitement automatique. Cette étape consiste à construire une représentation de la signification d'une expression (intent), par l'usage des sens de chaque mot la constituant. La difficulté de cette phase est amenée par la richesse du langage

6. http ://universaldependencies.org/u/pos/index.html
7. http ://universaldependencies.org/u/dep/index.html
8. https ://nlp.stanford.edu/pubs/USD_LREC14_paper_camera_ready.pdf
9. Utilisation d'analyses statistiques, probabilistes sur les données, pour l'obtention des résultats

naturel et les différentes ambiguïtés sémantiques qui s'y retrouves. En prenant par exemple l'expression conditionnelle "lorsque les habitants prennent un café". Il existe des ambiguïtés necessitant l'intervention humaine pour être interprétées. En effet, il n'est pas possible de savoir directement s'il s'agit d'un seul et même café pour tous les habitants, ou s'il s'agit d'un café chacun. Une autre ambiguïté existe avec le verbe "prennent", qui peut signifier que les habitant saisissent ce café, ou leur café respectif au même moment ou non, mais également une ambiguïté avec "lorsque" qui peut définir que l'action peut s'effectuer tout au long de la pause, ou au contraire juste au début de celle-ci. L'identification de ces *intent* se fait par apprentissage de corpus. L'outil LUIS proposé par Microsoft [10] est un des outils qui permettent cet apprentissage et son utilisation.
La chaîne de traitement de la solution [5] réalise en parallèle l'étape d'analyse sémantique avec l'outil LUIS et l'étape d'analyse syntaxique avec l'outil Bonsaï. Finalement, la fusion des deux résultats amènent à identifier les ambiguïtés qu'il est nécessaire d'expliciter avant de générer les règles techniques d'automatisation de l'habitat. Le feedback mis en place fait un bilan de ce que le système comprend et des propositions lorsque celà est possible, pour lever les ambiguïtés.

1.3.1 Limitations de l'approche

La solution [5] propose une chaîne de traitement itérative pour désambiguïser la langage naturel utilisé pour programmer son habitat. Cette solution présente des limitations dans sa mise en oeuvre.
Tout d'abord, **le "monde" considéré de l'habitat est simplifié** pour faciliter la désambiguïsation. Ainsi une seule instance de chaque objet ne peut être pris en compte (il n'est pas possible de considérer plusieurs télévisions par exemple). Ceci est dû à l'impossibilité d'associer aux objets des compléments de contextualisation (comme la localisation) lors de la chaîne de traitement. La seconde limitation de la solution [5] est la **complexité des règles OpenHab** pouvant être générées. En effet, cette solution ne peut générer que trois types de règles temporelles, d'états et d'actions. Chacune ne peuvent avoir qu'une seule condition simple et qu'une seule action.
Enfin les outils utilisés pour mettre en oeuvre cette solution ont rendu nécessaire la fusion des connaissances issues des analyses syntaxiques et sémantiques introduisant aussi la possibilité d'erreurs de traitement.
L'objectif de nos travaux est de rendre plus flexible et robuste la solution programmée par [5] en revoyant la chaîne de traitement avec l'utilisation de nouveaux outils pouvant la dé-complexifier, mais également en réalisant un langage supplémentaire (modèle "pivot") permettant de passer de la sémantique de l'habitat vue par l'habitant, à celle connue par OpenHab.

2 Une nouvelle chaîne de traitement

Pour palier les limitations exposées précedemment, nous avons modifié les outils utilisés pour les analyses syntaxiques et sémantiques et les traitements de normalisation et de génération des règles. L'étape d'analyse lexicale n'entrainait pas de limitations et la boucle de feedback n'a pas été étudiée dans le cadre de ces travaux. La nouvelle chaîne de traitement qui en résulte est présentée Figure 4. Le choix de modifier les outils utilisés pour l'analyse syntaxique (SyntaxNet) et pour l'analyse sémantique (Rasa_NLU) résulte de plusieurs tests sur les différents

10. https ://www.luis.ai/

outils proposés (repère des erreurs, resultats incohérents, formats, etc.) et de leur résultat afin d'arriver à une représentation machine finale du traitement effectué.
La phase de *normalisation* de la nouvelle chaîne, contrairement aux anciens travaux, traite uniquement la ponctuation. La ponctuation fausse l'étape d'analyse lexicale. Prenons par exemple la requête "À midi, éteins les lumières", sans la normalisation, nous obtenons une seule entité pour "midi," plutôt que "midi" et ",", ainsi elle n'est pas reconnue dans le traitement suivant. L'outil SyntaxNet utilisé pour l'étape d'analyse syntaxique ne dipose pas des problèmes rencontrés avec l'outil Bonsaï, qui survenaient avec la négation et les expressions impersonnelles par exemple. De plus, les synonymes ne faussent pas l'étape d'analyse lexicale réalisée avec ce nouvel outil, il n'est donc plus nécessaire de les prendre en compte dans la normalisation.
Nous proposons l'exécution des étapes d'analyse syntaxique et sémantique séquentiellement plutôt qu'en parallèle (comme le proposait [5]). En effet, il n'est pas nécessaire de réaliser ces deux étapes en même temps et cela ajoute de la complexité à la chaine de traitement. L'analyse syntaxique (étudiée avec l'outil SyntaxNet) permet ainsi d'obtenir les relations de dépendances entre les "tokens" (*ref.* Figure 3). Cette analyse est essentielle afin de savoir quels éléments ont une action sur quels autres. En effet, dans le cas d'une requête avec plusieurs objets et plusieurs actions, il est nécessaire de connaître quelles actions sont à réaliser sur quels objets présents dans cette même requête (en prenant par example "allume la télévision et éteins la lumière", l'analyse permet de savoir que l'action "allumer" porte sur l'objet "télévision" et que l'action "éteindre" porte sur l'objet "lumière").
Ensuite, l'analyse sémantique permet de donner un sens à la requête en la catégorisant avec l'un des plusieurs types pré-définis, tels que les expressions temporelles (expressions disposants de conditions sur la date et/ou le temps seulement), les expressions d'état (expressions disposants de conditions sur des états d'objets uniquement) et les expressions disposant des deux types précédants (expressions disposants d'une condition globale constituée de conditions temporelles et de conditions d'état). Elle permet d'autant plus de confirmer une partie du résultat retourné par l'étape précédante (validation des appareils trouvés dans l'analyse syntaxique par rapport à ceux trouvés dans l'analyse sémantique). Nous avons changé le logiciel responsable de cette étape afin de faciliter l'utilisation de la solution. Celle-ci est réalisée avec l'outil Rasa_NLU qui a pour premier avantage d'être Open Source et installable localement contrairement au service web LUIS de Microsoft, qui ne peut être utilisé dans un système n'ayant pas accès au réseau Internet. D'autant plus que le nombre de requêtes envoyées au service est borné et qu'il est nécessaire de payer afin de supprimer cette limite fixée. Rasa_NLU laisse la possibilité d'importer des données d'apprentissage provenant de LUIS, nous avons donc utilisé le corpus d'apprentissage de [5].
Enfin, un fois que le type de règle à générer est défini, que les noms des appareils et des actions sont trouvés et mis en relation, mais également que la différentiation de la partie conditionnelle, de la partie d'execution de la requête est identifiée, il ne reste plus qu'à lier ces connaissances aux données du système (l'habitat intelligent) pour générer les règles OpenHab correspondantes. Nous proposons d'utiliser un modèle "pivot" pour permettre le passage du langage naturel au langage informatisé.

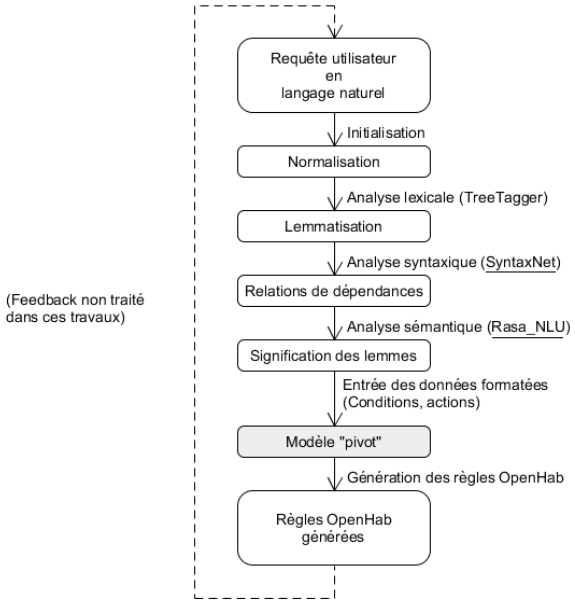


FIGURE 4 – Chaîne de traitement d'une requête formulée en langage naturel

3 Création et validation du modèle "pivot"

3.1 Le modèle

Le modèle "pivot" permet d'exprimer les concepts de l'habitat utile pour sa programmation (tels que les objets à contrôler), en s'abstrayant de la manière dont ils sont définis dans chacun des langages (ie : en langage informatique et en langage naturel). Ainsi, le concept de télévision est une instance d'un objet (Device) associé à des services (AssociatedService et Service) indépendamment du fait qu'OpenHab travaille sur des "items" qui définissent la plus basse granularité manipulable dans l'habitat, mais qui ne correspondent pas à des instances d'objets (au sens "humain" du terme).

3.2 Génération de règles OpenHab à partir du modèle "pivot"

Le modèle "pivot" a deux fonctions, adapter le traitement de la commande en langage naturel dans une configuration d'habitat particulière et de générer les règles OpenHab correspondantes. Pour permettre l'adaptation, il peuple sa représentation de l'habitat (pièces, appareils, services...) sur lequel il travaille avec un fichier de configuration spécial propre à chaque habitat (*ref.* Annexes - Exemple de données de configuration modèle "pivot"). Ce fichier donne également les informations techniques relatives et nécessaires pour pouvoir manipuler l'habitat (commandes et entités OpenHab etc.). Le fichier de configuration permet de faire le lien avec les classes d'objets du modèle "pivot" en créant des instances pour chaque élément technique

du fichier.
La génération de règles OpenHab repose sur l'instanciation des classes du modèle dédiées à l'expression de la condition et à l'expression des actions. Une instance de règle est réalisée grace aux différentes instances de classes créées précédemment. L'instance de l'expression de la condition est composée d'un arbre conditionnel binaire, dont les noeuds sont des liens "AND" et "OR" et les feuilles des expressions booléennes. Le nombre de règles nécessaire à générer pour répondre correctement à la demande analysée dépend de la structure de cet arbre. Bien que le modèle "pivot" permette la programmation répondant à des conditions composées, une instance d'abre binaire de conditions ne peut être composée que par des noeuds "OR" ou un unique noeud "AND" à la racine. Cette limite simplifie la génération pour prendre en compte une limitation d'OpenHab sur l'expressivité des conditions de déclenchement. Elle se lève par la génération de plusieurs règles dans le cas où la condition exprimée par l'habitant nécessite plusieurs "AND". Lorsque le modèle débute la génération de règles, il existe trois situations possibles pour l'abre conditionnel donné, celle où la racine de l'arbre est une feuille (arbre binaire de taille 1), celle où elle correspond à un opérateur conditionnel "OR" (arbre binaire de taille n, avec $n \geq 3$) et celle où il s'agit d'un opérateur "AND" (arbre binaire de même taille n). Dans les deux premières situations, la génération d'une seule règle est suffisante et peut être faite directement avec les données passées (nom, condition et actions). Dans le dernier cas, il est nécessaire d'en réaliser deux afin de répondre convenablement à la requête formulée. Ainsi pour répondre convenablement au résultat souhaité, nous générons une première règle en tenant compte du sous-arbre gauche comme arbre conditionnel de cette règle et du sous-arbre droit comme condition dans le bloc d'éxecutions.

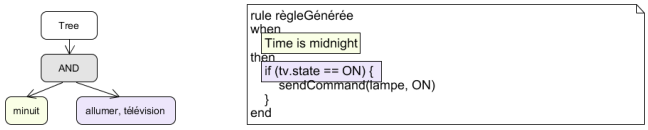


FIGURE 5 – Génération de règle simple - sous-arbre gauche

Inversement pour la règle complémentaire, nous utilisons le sous-arbre gauche comme condition pour le bloc d'éxecutions et le sous-arbre droit comme arbre conditionnel de la règle.

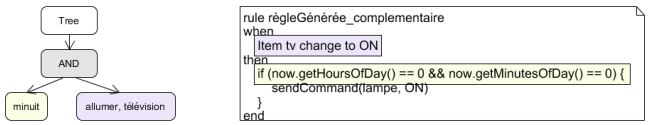
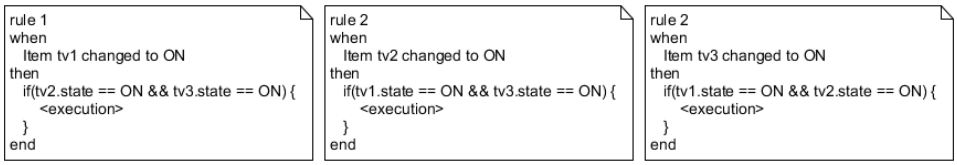


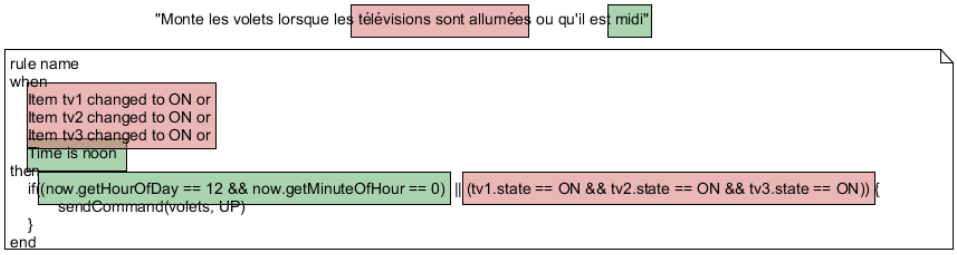
FIGURE 6 – Génération de règle complémentaire simple - sous-arbre droit

Une seule des deux règles dans le cas d'un opérateur de conjonction à la racine, ne permet pas de réaliser le besoin formulé par l'habitant. Il est donc nécessaire de prendre en compte les deux règles générées pour répondre totalement à ce besoin.
Dans toutes les situations, il est nécessaire de prendre en compte les objets du modèle représentant un ensemble d'entités OpenHab, mais également les objets multiples. Prenons comme

exemple un habitat dans lequel plusieurs télévisions et plusieurs lumières sont disponibles. L'expression "éteins les lumières, lorsque les télévisions sont allumées" peut être définie simplement au niveau de l'action à réaliser par un ensemble de commandes d'extinction (sendCommand(<item-activable-lumiere>, <commande-extinction>)), disposant d'autant de commandes qu'il y a d'appareils correspondants. Tandis que pour la condition "lorsque les télévisions sont allumées", il est obligatoire que toutes les télévisions ciblées soient allumées afin que la condition de la règle soit valide. Cette obligation se determine par une conjonction (par exemple, télévision du salon allumée **ET** télévision de la cuisine allumée), ce qui ne permet pas de l'insérer directement dans le bloc conditionnel de la règle à générer. Une première solution consiste à réaliser autant de règles qu'il existe de clauses (*ref.* Figure 7.a). Cependant cette approche peut demander la construction d'un nombre important de règles et ne fonctionne que si la condition se limite à cette même conjonction. Une seconde solution qui a été adoptée est de réaliser un premier cycle permettant de générer la ou les règles nécessaires et finalement d'ajouter une disjonction dans chaque bloc conditionnel et une conjonction ainsi que de la condition globale (condition donnée pour la génération de la règle) dans chaque bloc d'execution de ces règles (*ref.* Figure 7.b). Cette dernière modifie simplement une règle déjà générée afin de la rendre efficace, ce qui allège l'algorithme de génération employé par le modèle.
Tout au long de la génération des règles, le modèle peut à tout moment remonter une exception si un appareil, ou un service, ou encore une pièce recherchée de l'habitat n'a pas été trouvée, mais également si une erreur de conversion de données survient (par exemple pour la conversion d'une date ou d'un temps). Le modèle donne la possibilité de gérer certaines phases d'ambiguïté par l'utilisation d'une classe d'appel (fonctionnement équivalent à un "callback" avec retour). Cette classe permet la création et l'appel d'un élément externe au modèle afin de désambiguïser un resultat, avant de reprendre au même endroit la génération.



(a) Première solution pour la gestion des cas particuliers de conjonctions



(b) Solution adoptée pour la gestion des cas particuliers de conjonctions

FIGURE 7 – Exemple de règle OpenHab

3.3 Validation du modèle et de la génération

Le modèle a été conçu sous la forme d'une représentation en diagrammes de classes UML (*ref.* Annexes - Figures 8 et 9). Une preuve de concept a été réalisée et validée sous forme de tests unitaires, c'est à dire la génération d'une règle OpenHab à partir d'une expression analysée. Ces expressions sont issues du traitement d'un corpus de 42 commandes en langage naturel sur un habitat de simulation et sur des données d'un habitat réel (DOMUS [11]). Sur les 24 tests unitaires effectués, 17 produisent une règle OpenHab correcte, 4 permettent de detecter des erreurs de construction de la requête (ex : commande sur un appareil inexistant dans l'habitat) et 3 illustrent des limitations de notre approche par rapport à la puissance d'expression d'OpenHab. Cette preuve permet en plus d'obtenir l'ensemble des retours possibles en fonctions des règles à réaliser, pour les prendre en compte dans l'interaction avec l'utilisateur dans des travaux futurs.
Le modèle a également été mis en place en tenant compte des différentes données nécessaires à la génération complète d'une règle OpenHab cohérente pour l'expression analysée (la condition complexe de lancement de la règle et les actions a éxécuter). De plus, ses limites ont été évaluées dans sa version acutelle afin de connaître les éléments techniques OpenHab (version 1.X) pris en charge.
La génération a été testée sur des données (lemmes définissant des appareils et des actions) provenant des travaux précédents. Ces données ont été employées dans le modèle afin de générer des règles qui ont ensuite été validées syntaxiquement avec le logiciel OpenHab Designer [12] et sémantiquement avec un serveur de simulation OpenHab.

4 Conclusion & futurs travaux

Nous nous sommes interessés dans ce document à la compréhension des expressions en langage naturel formulées par les habitants, afin de générer les règles d'automatisation de leur habitat. Les travaux précédents permettent de générer des règles OpenHab simples disposant de peu d'instructions. En comparaison, la nouvelle version propose un nouveau langage permettant d'abstraire le contexte technique du foyer, mais également de réaliser des règles plus complexes pouvant mieux satisfaire les besoins des habitants. En plus de celà, ce nouveau langage offre la possibilité de lui ajouter plus facilement des extensions et de prendre en compte les perspectives proposées par les travaux précédants, telles que les instances multiples d'objets et leur localisation dans l'habitat. Cette version propose en outre une nouvelle chaîne de traitement du langage naturel, moins complexe et utilisant des outils plus récents et plus libres d'utilisation.
Les travaux réalisés proposent plusieurs perspectives interessantes. Une première serait d'implémenter la nouvelle chaîne de traitement, en employant le modèle réalisé pour la génération des règles techniques et de tester la solution obtenue dans un véritable environnement tel que DOMUS. Une seconde serait de réaliser une extension des conversions prises en compte par le modèle "pivot" afin d'ajouter de la flexibilité au modèle (par exemple la conversion du temps donné par l'habitant en tenant compte des jours, mois et années, contrairement au heures, minutes et secondes seulements prises en charge actuellement).
Enfin, l'usage d'un dialogue en temps réel identiquement à un système de chat, ou SMS pourrait être interessant pour désambiguïser l'expression formulée et permettant également de réaliser un suivit naturel de l'utilisateur dans sa démarche de création d'automatismes pour son habitat.

11. http ://domus.liglab.fr/
12. http ://docs.openhab.org/installation/designer.html

5 Références

[1] Pierre SARRAT, *Le projet Français Habitat Intelligent/Domotique et le SED*, Actes du colloque, Tome I, 1989.
[2] Pierre BRUN, *La Domotique*, Que sais-je, 1988.
[3] Henry LIEBERMAN & Fabio PATERNÓ & Markus KLANN & Volker WULF, *End-User Development : An Emerging Paradigm*, 2-3.
[4] Emeric FONTAINE & Alexandre DEMEURE & Joëlle COUTAZ & Nadine MANDRAN, *Retour d'expérience sur KISS, un outil de développement d'habitat intelligent par l'utilisateur final*, IHM 2012, ACM, 2012, 4-6.
[5] Lorrie ROUILLAUX, *Dialoguer avec un habitat intelligent*, 2016.

6 Annexes

6.1 Diagrammes de classes modèle "pivot"

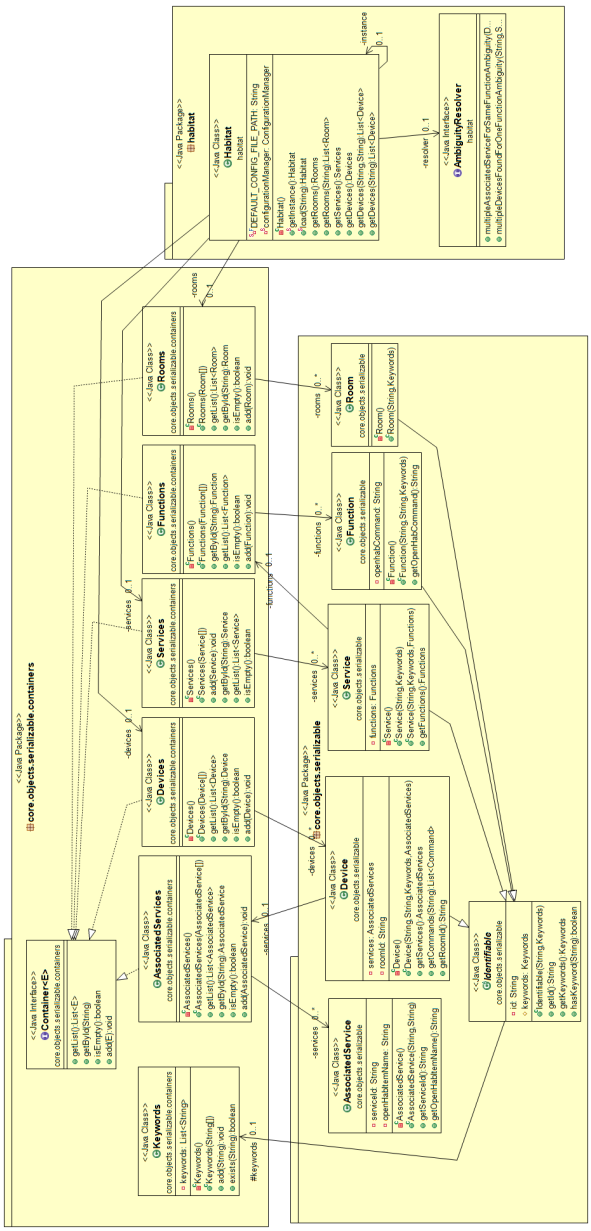


FIGURE 8 – Diagramme de classes UML - Entités de modélisation d'habitat du modèle "pivot"

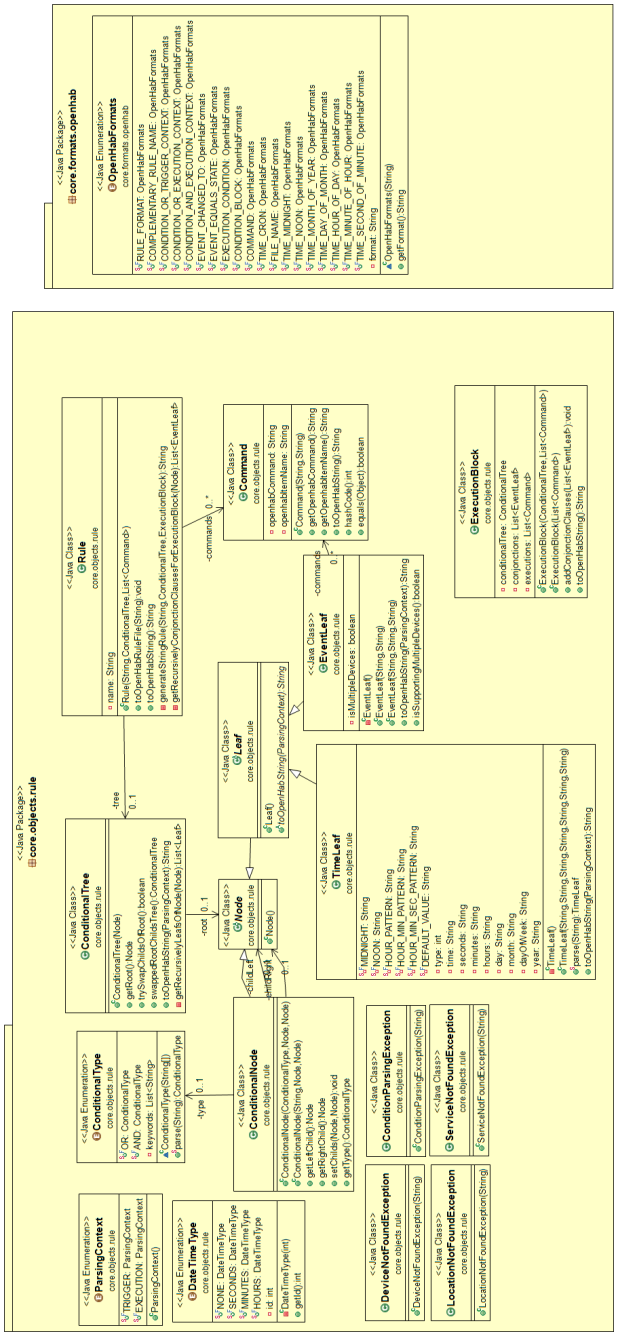


FIGURE 9 – Diagramme de classes UML - Package permettant la génération de règles OpenHab du modèle "pivot"

6.2 Exemple de données de configuration modèle "pivot"

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<habitat>
    <rooms>
        <room id="cuisine">
            <keywords>
                <keyword>cuisine</keyword>
            </keywords>
        </room>
        <room id="salon">
            <keywords>
                <keyword>salle a manger</keyword>
                <keyword>salon</keyword>
            </keywords>
        </room>
    </rooms>
    <services>
        <service id="activable">
            <keywords>
                <keyword>activer</keyword>
                <keyword>activable</keyword>
            </keywords>
            <functions>
                <function id="allumer" command="ON">
                    <keywords>
                        <keyword>activer</keyword>
                        <keyword>allumer</keyword>
                    </keywords>
                </function>
                <function id="eteindre" command="OFF">
                    <keywords>
                        <keyword>eteindre</keyword>
                        <keyword>arreter</keyword>
                    </keywords>
                </function>
            </functions>
        </service>
    </services>
    <devices>
        <device id="lampe" room="cuisine">
            <keywords>
                <keyword>lumiere</keyword>
                <keyword>lampe</keyword>
            </keywords>
            <services>
                <service id="activable" item="lampe_power"/>
            </services>
        </device>
        <device id="television" room="salon">
            <keywords>
                <keyword>tv</keyword>
                <keyword>tele</keyword>
                <keyword>television</keyword>
            </keywords>
            <services>
                <service id="activable" item="tv_power"/>
            </services>
        </device>
    </devices>
</habitat>
```