Modélisation de l'interopérabilité d'objets communicants et de leur coopération : application à la domotique

Fabien Sartor

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **STIC Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Fabien Sartor

Thèse dirigée par **Patrice Moreaux**
et codirigée par **Flavien Vernier**

préparée au sein **LISTIC**
et de **SISEO**

Modélisation de l'interopérabilité d'objets communicants et de leur coopération

Application à la domotique

Thèse soutenue publiquement le **jeudi 5 juillet 2012**,
devant le jury composé de :

Fulvio Corno
Professeur associé (Polytechnico di Torino), Rapporteur
Pierre Gerinière
Directeur innovation technologique de SOMFY, Examinateur
Attilo Giordana
Professeur (Università del Piemonte Orientale « A Avogadro »), Examinateur
Jean-Noël Loiseau
Directeur de Overkiz, Examinateur
Éric Niel
Professeur (INSA de Lyon), Rapporteur
Kavé Salamatian
Professeur (Université de Savoie), Président
Patrice Moreaux
Professeur (Université de Savoie), Directeur de thèse
Flavien Vernier
Maître de conférence (Université de Savioe), Co-Directeur de thèse

Université Joseph Fourier / Université Pierre Mendès France /
Université Stendhal / Université de Savoie / Grenoble INP

Résumé

Dans le cadre des évolutions du bâtiment, il est de plus en plus nécessaire d'interconnecter des objets communicants. Cette démarche est cependant freinée du fait de l'absence d'un protocole de communication standard. La maison intelligente n'est pas un nouveau concept, et l'automatisation de la maison est devenue un sujet de recherche à la mode ces dernières années.

La problématique de cette thèse se focalise sur l'interopérabilité de systèmes communicants. Plus précisément, nous nous intéressons à la manière de créer une coopération entre différents dispositifs d'un environnement, afin de masquer la complexité à l'usager.

Dans un premier temps, l'interopérabilité entre ces objets est réalisée par la mise en place d'un écosystème virtuel où les dispositifs peuvent communiquer leurs états ou l'état de l'environnement. C'est l'abstraction des systèmes. Dans ce mémoire, nous abordons comment et pourquoi les systèmes sont nécessairement abstraits lorsque l'on s'intéresse à la prise en compte du contexte de l'installation. L'étude de la prise en compte du contexte à permis de procurer les données scientifiques à l'entreprise Overkiz afin qu'elle puisse réaliser l'abstraction des dispositifs domotiques.

Ensuite, la coopération entre les objets communicants est mise en œuvre par un outil mathématique permettant de modéliser le comportement d'un environnement composé d'actionneurs, de capteurs et d'interfaces utilisateurs. Le comportement est formalisé au moyen de la théorie des automates étendus et plus particulièrement la théorie des systèmes de transitions symboliques à entrées/sorties (IOSTS). Nous synthétisons alors le pilotage par un « contrôleur » du comportement global d'une installation, à partir de règles de contraintes ou de règles d'action.

mots-clé : automates étendus, comportement, domotique, ontologies

Abstract

In the context of the building developments, it is increasing necessary to interconnect communicating objects. This approach is slow down because of the lack of a standard communication protocol. The smart home is not a new concept, and home automation has become a fashion research topic in recent years.

The problem of this thesis focuses on the interoperability of communicating systems. Specifically, we focus on how to establish cooperation between different devices of an environment in order to hide the complexity to a user.

Initially, interoperability between these objects is achieved by the establishment of a virtual ecosystem where devices can communicate their states or the state of the environment. It is the abstraction of systems. In this paper, we discuss how and why systems are necessarily abstract when we are interested in taking into consideration the context of the installation. The study of consideration of context allowed to provide scientific data to the company Overkiz in order to realize abstraction of home automation devices.

Second, cooperation among communicating objects is implemented by a mathematical tool to model the behavior of an environment cibsustubg of actuators, sensors and user interfaces. The behavior is formalized using the automata theory and more specificaly the automaton theory of Input/Output Symbolic Transition Systems (IOSTS). We synthesize then how to synthetise a "controller" of a global behavior of an installation, from constraint rules or rules of action.

mots-clé : extended automaton, behaviour, home automation, ontologies

Remerciements

Je souhaiterais tout d'abord remercier mes encadrants de thèse, Patrice Moreaux, Flavien Vernier et Laurent Zilber, pour l'encadrement de mon travail, leurs conseils avisés et leurs remarques pertinentes. Ils m'ont permis d'améliorer la qualité de ma thèse et de bien cerner toute la problématique.

Merci également aux personnes qui m'ont fait l'honneur d'accepter de participer à mon jury, Fulvio Corno, professeur associé à Polytechnico di Torino, Pierre Gerinière, directeur innovation technologique de l'entreprise SOMFY, Attilo Giordana, professeur à l'université de Piemonte Orientale à Avogadro, Jean-Noël Loiseau, directeur de l'entreprise Overkiz, Éric Niel, professeur à l'INSA de Lyon, Kavé Salamatian, professeur de l'Université de Savoie. Merci à Dario Bonino, assistant à Polytechnico di Torino, d'avoir fait le déplacement depuis Turin pour assister à ma soutenance.

Ces travaux n'auraient ni abouti ni même débuté sans le soutien de l'entreprise Overkiz et de ses collaborateurs : Jean-Noël, merci pour ces moments de management, de discussions amicales et d'humours. Silvain, merci pour tes conseils techniques, tes histoires toujours détaillées et la beauté de ton code. Boris, Julien et Gael, merci pour votre amitié et votre écoute. Merci au reste du personnel d'Overkiz de m'avoir si bien accueilli.

Je remercie tous les chercheurs, enseignants, membres du personnel du laboratoire LISTIC pour leur amitié et leur aide pendant ces trois années de thèse. Merci à Florent, Jean-Claude, Joëlle, Maz, Philippe et Sylvie. Un grand merci à mes amis doctorants, Adbellah, Renaud, Aziz (El Hachemi), Tiberius et Liviu, pour votre soutien et le divertissement que vous m'avez procuré.

Je remercie mes amis de l'école d'ingénieur, du canada et de l'IUT pour les bons moments que l'on a vécus ensemble. Merci aux membres de l'association d'Échanges Sahel pour le temps et le dévouement qu'ils consacrent à une cause que j'estime.

Merci à Aurore qui a partagé ma vie, mes moments d'enthousiasme et mes doutes durant ces trois années de recherche.

Enfin, je souhaiterais remercier toute ma famille pour leur patience et leur soutien. Merci à mes parents et ma sœur de m'avoir aidé à garder l'équilibre et poussé dans l'accomplissement de mon travail.

À mon grand-père.

Table des matières

Table des figures

Introduction

Vingt ans après que Mark Weiser ait exposé sa vision de l'informatique pour le XXI^{eme} siècle [85], nous commençons à avoir des solutions pour « l'informatique diffuse » économiquement viables. Les progrès de la technologie sont énormes : il est possible de consulter ses courriers électroniques depuis son téléphone portable, ou même de regarder la télévision dans le train. L'évolution des usages amène à de nouveaux comportements et de nouveaux besoins.

Le concept d'habitat existe depuis la nuit des temps. Le rapport de l'homme à sa maison évolue lentement et met en jeu des notions culturelles, sociales et de confort, en plus de la protection évidente qu'apporte l'habitat [62]. La domotique laisse entrevoir une évolution de ces notions en les enrichissant de nouvelles commodités mises à disposition de l'homme.

D'un point de vu industriel, ces travaux de thèse s'inscrivent dans le cadre de l'amélioration d'une solution développée par l'entreprise Overkiz. Cette solution permet la gestion d'une installation domotique. Plus précisément, elle propose des téléservices dans le domaine de la maison. Ces téléservices sont proposés à différents acteurs comme les habitants d'une installation, les installateurs d'équipements ou encore les constructeurs d'appareils pour la maison.

Dans le cadre des évolutions du bâtiment, il est de plus en plus nécessaire d'interconnecter ces différents dispositifs. Cette démarche est cependant freinée du fait de l'absence d'un protocole de communication standard. En effet depuis quelques années, on assiste à une profusion de protocoles soit propriétaires, soit ouverts.

Contexte

La maison intelligente n'est pas un nouveau concept, et l'automatisation de la maison est devenue un sujet de recherche à la mode ces dernières années. Cette évolution est partiellement due à la prise de conscience collective face aux changements climatiques et au fait que le bâtiment est un des secteurs les plus énergivores. À titre d'exemple, le secteur "résidentiel et tertiaire" [1] est le plus gros consommateur d'énergie en France et représente 44% de l'énergie consommée en 2007 [14].

Cependant, la maison intelligente ne se limite pas à la gestion de l'énergie ; elle est indispensable pour améliorer le confort et la sécurité de l'utilisateur. Avec le vieillissement de la population en Europe occidentale, le maintien à domicile des personnes handicapées et âgées est une nécessité. Dans ce sens, la domotique apporte une solution et améliore le confort des personnes dépendantes. Une autre promesse de la maison intelligente est la sécurité des biens et des personnes. Une installation domotique munie de capteurs permettra d'alerter à distance un usager d'une intrusion, d'un fonctionnement anormal ou même d'un incendie.

Vers une informatique omniprésente

De nos jours, l'intégration de systèmes informatiques dans l'environnement réel est basée sur le paradigme *person-to-computer* ou *machine-to-machine*. Le paradigme *person-to-computer* est centré utilisateur. Dans le domaine de la domotique, il s'agit principalement de dispositifs coûteux comme l'électroménager, les ouvrants ou les appareils de communication. À l'opposé, le paradigme *machine-to-machine* est centré environnement physique. Dans ce contexte, l'informatique est utilisée pour contrôler un processus. Par exemple, un centre de tri automatique de courrier postal utilise plusieurs systèmes communicants.

Les progrès des technologies de communication informatique et le développement de la puissance de calcul font apparaître de nouvelles opportunités pour l'informatique. Grâce à la réduction continue des coûts de fabrication et de la taille des composants, il est maintenant possible de rendre tous les objets communicants. On parle de systèmes ubiquitaires. Le terme « Informatique ubiquitaire » est apparu suite à un article de Mark Weiser qui, en 1995, présente sa vision sur l'intégration des calculateurs dans l'environnement[85].

L'objectif de l'informatique ubiquitaire est de réaliser l'intégration transparente pour des utilisateurs d'un environnement numérique dans l'environnement réel. L'informatique ubiquitaire ou intelligence ambiante doit offrir aux usagés un mode d'interaction naturel avec le monde réel. À terme, les utilisateurs n'auront

1. Habitat, bureaux, locaux collectifs, etc.

plus conscience d'interagir avec des calculateurs. L'intelligence ambiante pousse le paradigme *person-to-computer* vers un nouveau concept : *person to physical world* [76].

Pour atteindre cet objectif, l'aspect le plus important est la modélisation du monde réel dans un espace virtuel. Les objets d'un environnement doivent être capables de transmettre l'information à des systèmes plus complexes et de récupérer de l'information d'autres systèmes (notion de communication). Les objets communicants ont différentes utilités ; ils peuvent caractériser l'environnement (notion de capteur), agir sur l'environnement (notion d'actionneur) et/ou échanger de l'information avec un utilisateur (notion d'interface).

Dans un avenir proche, les objets communicants vont s'intégrer dans notre vie de tous les jours. Ils collecteront et traiteront des informations de différentes sources et interagiront avec les utilisateurs. La notion de communication est la base de l'intelligence ambiante. Elle permet aux éléments calculatoires de collaborer et de fournir une représentation de l'environnement de l'utilisateur.

Cette communication crée une informatique ambiante qui, liée avec Internet, offrira de nouvelles possibilités et un Internet omniprésent.

Dans ce sens, un article propose de convertir les dispositifs de la maison en ressources disponibles sur Internet [40]. Les auteurs vont même plus loin et proposent de relier les prises électriques communicantes d'une installation aux réseaux sociaux [39]. Les prises permettent de relever la consommation électrique des appareils branchés. Ainsi, on peut partager et comparer la consommation énergétique d'appareils avec celle d'autres utilisateurs. De plus, il est possible d'allumer ou d'éteindre les appareils connectés aux prises depuis Internet. L'utilisation des réseaux sociaux permet ici d'utiliser les structures sociales existantes sur Internet. Les auteurs s'abstraient donc de la gestion des utilisateurs du système.

De nouvelles utilisations des équipements

Internet se développant, les frontières avec le mode réel s'effacent. Aujourd'hui, les équipements permettent d'accéder très facilement à des applications réparties. Ces équipements sont rendus de plus en plus portables et nomades et l'accès aux applications se fait de n'importe où, sans contrainte géographique.

Sur Internet, les usagers exécutent une multitude d'applications et utilisent un grand nombre de services. Des applications variées comme des jeux en réseau, la visualisation cartographique tel que géoportail [47], ou même un bureau virtuel comme eyeOS [22] sont disponibles. Les services dédiés utilisateurs ne sont pas en reste : envoi de mails, consultation d'encyclopédies, discussions instantanées...

Prenons pour exemple l'achat de biens. Les emplettes peuvent se faire sur Internet et sont livrées le lendemain directement chez l'utilisateur. Inversement, lors d'achats chez un commerçant l'accès à Internet avec un smartphone permet de comparer les prix avec les concurrents.

Les nouveaux équipements nomades permettent d'interagir avec des services distants, mais aussi des services ambiants. Les services distants sont accessibles partout et à tout instant. À l'inverse, les services ambiants sont locaux à un environnement, c'est-à-dire qu'ils sont disponibles dans un certain milieu et ces services changent lorsque l'on change d'environnement. Ces services peuvent être accessibles par différents intergiciels et utiliser différents protocoles de découverte (UPnP, Bonjour, etc.). L'évolution de l'environnement pour un équipement mobile pose la problématique des applicatifs adaptables à l'environnement [30].

De plus, de nouvelles perspectives s'affichent avec l'arrivée d'IPv6. En effet, l'intégration des objets communicants dans l'environnement permettra de relier le monde virtuel et le monde réel. En utilisant IPv6, chaque dispositif est atteignable sans se soucier de sa localisation géographique [46].

Problématique

Dans le cadre des évolutions du bâtiment, il est de plus en plus nécessaire de connecter les actionneurs avec des organes de commande interopérants. Les différentes solutions domotiques nécessitent d'agir sur un environnement et de recevoir des informations. L'origine de cette thèse est le besoin, pour l'entreprise Overkiz, d'interconnecter des objets de la maison. La problématique se focalise sur l'interopérabilité de systèmes communicants. Plus précisément, nous nous intéressons à la manière de créer une coopération entre différents objets communicants d'un environnement, afin de masquer la complexité à l'usager.

Cette thèse étudie une méthode permettant de créer un comportement d'un environnement. Plus particulièrement appliqué à un environnement domotique. Une telle approche a de nombreux objectifs :

- créer un environnement interconnecté afin de communiquer avec tous les dispositifs de manière transparente ;
- modéliser le comportement de systèmes ;
- modéliser le comportement d'un environnement composé de multiples systèmes indépendants les uns des autres ;
- fournir un moyen d'agir sur le comportement des systèmes d'un environnement.

Nous verrons que de nombreux travaux s'occupent de fournir des outils afin d'offrir un environnement interconnecté. Cependant, ces travaux ne s'occupent que de l'interopérabilité de systèmes réels et des services. De façon orthogonale, les travaux de cette thèse se focalisent sur le comportement des systèmes.

Plan du document

Outre l'introduction et la synthèse globale, ce mémoire de thèse est composé de deux parties.

La première partie est une présentation en trois chapitres de l'état de l'art sur les solutions domotiques, les solutions d'interconnexion et la modélisation du comportement.

Le chapitre 1 compare quelques travaux existants autour des solutions d'environnement ubiquitaire. Il rappelle les aspects essentiels des solutions puis il compare sept projets au moyen de ces aspects. De plus, il note comment les ontologies sont utilisées pour atteindre l'objectif d'interconnexion des systèmes. Il met en évidence le manque de travaux sur la gestion du comportement de tels systèmes.

Le chapitre 2 présente les moyens de gestion de la connaissance à l'aide d'ontologies. Il explique les concepts utilisés par les ontologies. Il présente différents types d'utilisation ainsi que certains outils permettant d'exploiter les ontologies.

Le chapitre 3 introduit le type de modélisation du comportement que nous avons retenu (*Input Output Symbolic Transition Systems*, IOSTS). Il rappelle les propriétés du modèle choisi et présente les outils permettant de l'exploiter.

La seconde partie de ce document concerne la problématique et la contribution apportée par les travaux de cette thèse. Elle est composée de quatre chapitres.

Le chapitre 4 explique comment est réalisée l'abstraction des objets communicants avec les ontologies. Ces mécanismes ont été mis en œuvre au sein de la solution Overkiz.

Le chapitre 5 définit le modèle de comportement appliqué aux objets communicants du monde de la maison. Cette modélisation que nous proposons se base sur une variante du modèle fondamental IOSTS. Dans cette adaptation des automates IOSTS, nous gérons, en plus de caractéristiques standards des IOSTS, l'absence de connaissance de l'état du système, c'est-à-dire les phases où le système est potentiellement en changement d'état.

Dans le chapitre 6, nous introduisons les mécanismes que nous proposons. Ils permettent la génération de comportements satisfaisants de l'environnement domotique. Nous montrons comment, à partir d'une connaissance sur les comportements possibles des éléments de l'installation et de règles de comportement, il est possible de synthétiser un contrôleur qui garantisse le comportement voulu pour

l'installation. Pour ce faire, nous proposons un ensemble d'opérations mathématiques de base sur les automates IOSTS ainsi que des algorithmes de synthèse des contrôleurs.

Le chapitre 7 décrit la mise en œuvre des méthodes et algorithmes à l'aide d'un exemple détaillé. Ce chapitre présente également, l'architecture de notre implantation et les outils logiciels sur lesquels nous nous appuyons.

I

Contexte et problématique

Sommaire

Chapitre 1

Les solutions domotiques

Le but de la domotique est de fournir une solution permettant de « domestiquer » une habitation. Les solutions de domotisation disponibles aujourd'hui sont conçues au moyen de composants traditionnels et de leur évolution. Elles sont créées par des fabricants d'éléments de la maison ou des intégrateurs. Les premiers fournissent des solutions domotiques partielles avec pour chacun d'entre eux une couverture fonctionnelle différente et une politique mercatique différente. Les seconds proposent des solutions complètes combinant les solutions partielles dont le coût économique n'est pas accessible par le plus grand nombre d'utilisateurs. La banalisation de la domotique est freinée par plusieurs facteurs :

- le coût économique de la mise en place d'une solution complète,
- le manque de coopération entre les dispositifs,
- la faible couverture fonctionnelle des solutions.

Par exemple, le consortium Konnex (KNX) propose plus d'une centaine de dispositifs interopérables. Mais, l'installation de l'infrastructure de communication KNX au sein d'une habitation nécessite un investissement économique important. Ce coût est dû à mise en place du bus de communication, seul moyen pour faire communiquer les dispositifs entre eux. De plus, l'évolutivité d'une installation équipée d'un bus KNX est contrainte par l'infrastructure de câbles existants.

D'autres moyens d'accès aux dispositifs de la maison sont développés afin de rendre la mise en place d'une solution domotique plus accessible. Pour cela, ils

utilisent les éléments de bases constituant une habitation comme les lignes électriques ou l'air. Ces méthodes d'accès sont plus flexibles et sont plus faciles à installer, car elles n'impliquent pas de manipulation du média de communication. Par exemple, la technologie du courant porteur en ligne transforme les informations en un signal électrique et superpose ce dernier au courant électrique distribué par les lignes électriques.

Le terme interopérable, appliqué à des dispositifs, signifie que les dispositifs possèdent un repère sémantique permettant d'activer des opérations ou d'acquérir des informations sur l'environnement.

Il existe une longue liste d'écosystèmes disponibles permettant de domotiser une habitation. Ce marché est très actif et de nouvelles technologies apparaissent continuellement. L'hétérogénéité des écosystèmes provient à la fois des différences techniques mises en place pour créer l'infrastructure de communication et de la couverture fonctionnelle de ces écosystèmes. L'utilisation conjointe de plusieurs écosystèmes permet de combler les manques fonctionnels. Mais, l'incompatibilité entre les écosystèmes empêche la coopération entre les dispositifs, augmentant par la même occasion le nombre de points de contrôle que l'utilisateur doit manipuler pour faire fonctionner son habitation. Si l'utilisation conjointe d'écosystèmes est viable pour des habitations disposant de quelques dizaines de dispositifs, elle devient un problème inacceptable pour des installations plus grosses comme les hôpitaux, les hôtels ou les futures installations domotiques.

Un deuxième problème est l'évolution du nombre et du niveau de fonctionnalités proposées par les solutions domotiques. En effet, les utilisateurs vont développer de nouvelles demandes en même temps qu'ils utiliseront les fonctionnalités disponibles de leur habitation automatisée. L'une des attentes de ces fonctionnalités est de prendre en compte le contexte de l'installation afin d'adapter leur comportement sur l'environnement. Le contexte de l'installation est l'ensemble des informations utiles afin d'effectuer un choix automatique ou semi-automatique lors de l'utilisation d'une fonctionnalité. L'auteur de l'article [18] indique que les informations utiles pour décrire le contexte d'un système font références aux informations du monde réel, mais aussi aux informations relatives aux autres fonctionnalités utilisables pour le système (envoie de SMS par exemple).

Les solutions de domotisation de l'habitat doivent par ailleurs prendre en compte l'interaction multimodale. En effet, les fonctionnalités disponibles dans une installation sont accessibles aux utilisateurs depuis différentes interfaces, ces dernières sont distribuées dans les écosystèmes ou accessibles depuis des services.

1.1 Méthode d'analyse

Rappelons qu'à la différence des logiciels purement applicatifs, les solutions d'environnement intelligent ont pour objectif d'interagir avec le monde réel. Ces solutions sont composées de services fonctionnels et utilisent une base commune d'éléments logiciels non fonctionnels. Ces types d'application font face à des besoins communs tel que :

- l'utilisation d'un modèle conceptuel afin de « comprendre » l'environnement,
- l'abstraction des dispositifs pour utiliser les capteurs et les actionneurs aux fonctionnalités hétérogènes,
- un intergiciel pour utiliser des médias et des plate-formes hétérogènes,
- la prise en compte du contexte et de son évolution.

Ces besoins sont sélectionnés parmi les articles [64, 67, 9]. Afin de régler ces problèmes de manière globale et générique, un ensemble d'outils doit être fourni. Ces outils peuvent ne pas tous être disponibles sur une même plate-forme d'exécution et doivent communiquer entre eux ; par exemple, l'interface d'une application s'exécutant sur un téléphone mobile doit communiquer une commande à un dispositif de la maison. Cette communication entre les outils est réalisée par l'intermédiaire d'une application transverse appelée intergiciel. L'intergiciel et les éléments logiciels non fonctionnels doivent fournir un framework qui satisfasse les besoins communs des solutions d'environnement « intelligent ».

1.1.1 Modélisation conceptuelle

La phase de modélisation conceptuelle est l'une des étapes préliminaires au développement d'une application. Le modèle conceptuel permet de représenter, éventuellement graphiquement, un ensemble de concepts liés sémantiquement entre eux, les relations entre les concepts étant décrites par des termes. Le modèle conceptuel appliqué à la conception d'une application permet de décrire les processus logiques régissant le fonctionnement de l'application. Appliqué à la modélisation des données, le modèle conceptuel permet de représenter la hiérarchie des types de données, les caractéristiques des données et la nature des relations. Une modélisation est une forme de communication entre personnes alors que la spécification d'un modèle tend à être plus formelle et fait appel à la rigueur des mathématiques. Il existe plusieurs normes permettant de décrire un modèle conceptuel tel que :

- UML pour la conception d'une application,
- MERISE pour la représentation des données...

D'autres normes permettent de spécifier un modèle conceptuel, par exemple LePUS3 décrit dans l'article [25].

Le dimensionnement de l'architecture d'un système est important afin de fournir les bases d'une application. Les solutions de domotisation sont des systèmes complexes faisant intervenir de nombreux concepts comme l'adaptation, la flexibilité, l'interopérabilité et la modularité. Peu de solutions fournissent une architecture standard et adaptée aux caractéristiques de l'environnement intelligent, la plupart des solutions se concentrent sur un environnement contraint et non évolutif.

1.1.2 Abstraction du matériel

Les techniques et algorithmes complexes mis en place dans les services fonctionnels utilisent les données provenant du monde réel et élaborent des actions sur ce monde. Les données sont fournies par les capteurs et les actions sont réalisées par des actionneurs déployés dans le monde réel. L'hétérogénéité des fonctionnalités des dispositifs et la disparité des médias d'accès engendrent un ensemble de problèmes tels que :

- la compatibilité des services lors de l'ajout d'un nouvel écosystème,
- le risque de dépendre de fonctionnalités trop spécifiques,
- la possibilité de réécrire une même portion de code pour l'accès aux dispositifs,
- ...

Les services de haut niveau doivent posséder les moyens de régler ces problèmes sans pour autant être spécifiques à classe de dispositifs ou spécifique à un média de communication. À cette fin, un élément logiciel doit être mis en œuvre afin de prendre en compte ces préoccupations. L'abstraction du matériel prend en charge les spécificités de ces dispositifs afin de rendre générique le développement des services. Tous les projets présentés dans ce mémoire possèdent un élément qui permet de faire l'abstraction des dispositifs. Il existe deux catégories de solution : la première est figée par le modèle de représentation des fonctionnalités des dispositifs, par exemple le projet Pise [7] et la seconde catégorie exploite une description sémantique des fonctionnalités, par exemple avec l'utilisation d'une ontologie.

1.1.3 Intergiciel

L'utilisation de différentes technologies propres aux applications domotiques implique un développement modulaire. Ce développement modulaire et le déploiement sur différentes architectures sont facilités par la mise à disposition d'une

infrastructure de communication appelée intergiciel. Un intergiciel, en anglais *middleware*, est un système logiciel visant à rendre transparent la répartition des différents composants d'une application. Il est caractérisé par les environnements sur lesquels il peut être utilisé et par les principaux outils de gestions des modules qui le composent.

Un intergiciel dédié à la domotique peut être caractérisé par : ses méthodes de répartition, sa compatibilité avec les plate-formes, ses techniques de modularité et ses techniques de gestion de l'application.

Répartition

L'environnement que commande une application de domotisation est évolutif :

- des dispositifs aux nouvelles fonctionnalités apparaissent continuellement de même que de nouvelles technologies de communication,
- le nombre de services proposés est amené à augmenter,
- les fonctionnalités des services sont fréquemment améliorées.

Pour rendre l'application évolutive, il faut que de nouvelles fonctionnalités puissent être intégrées sans remettre en cause les développements précédents. Pour cela, il faut fournir un environnement d'exécution qui facilite le découpage de l'application en modules et des outils de communication entre les modules. Lorsque les modules ont un comportement autonome, c'est-à-dire une « vie », ils sont nommés compostants.

Le type de répartition indique la technologie qui supporte l'exécution et la mise en relation des composants de l'application.

Comptabilité et modularité

Lorsqu'une application est repartie, ses composants sont déployés sur différentes plate-formes. Ces plate-formes peuvent être des environnements uniquement logiciels : Unix, Windows, JVM... ou des environnements dépendant du matériels : téléphones mobiles, ordinateurs personnels, tablettes tactiles... Le critère de compatibilité indique quelles sont les plate-formes compatibles avec l'application. Le critère de modularité indique quelles sont les technologies de communication utilisées.

Gestion des composants et gestion de l'application

Les solutions de domotisation de l'environnement sont des applications complexes, construites autour de plusieurs composants. La fiabilité d'une telle application se fonde sur la fiabilité des composants qui la constituent ainsi que sur la gestion de ces composants. Si un composant vient à tomber en panne, une application utilisant ce composant doit trouver des alternatives et éventuellement reconfigurer les autres

composants. Par exemple, si le composant d'envoi de SMS est en panne à cause d'une coupure du service, l'application pourra reconfigurer un composant d'alerte d'intrusions dans l'environnement afin d'utiliser l'envoi de courriers électroniques.

Cette reconfiguration nécessite des éléments particuliers qui permettent de reconfigurer en fonctionnement l'application et les composants, de détecter les pannes logicielles et matérielles et de chercher de nouveaux composants proposant des fonctionnalités similaires. Au niveau de la gestion des composants de l'application, l'un des exemples les plus aboutis est probablement le système Persona [80]. Une description de ce projet est disponible en section 1.5.3.

La reconfiguration de l'application lors de l'exécution est souvent offerte par le cadre logiciel qui l'accueille. C'est le cas du système OSGi. Ce framework permet de gérer le cycle de vie des composants. Une description plus approfondie est présentée en section 1.2.3.

1.1.4 Prise en compte du contexte

Les applications fournissant un environnement intelligent ont besoin de se référer à des situations élaborées. Une situation, ou contexte, est l'état de l'environnement à un moment donné. La définition du terme contexte ne fait pas consensus ; dans la suite de ce mémoire, nous utiliserons ce terme tel qu'il est défini dans l'article [18] :

> *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.*

Le contexte comprend toute information qui caractérise la situation d'une entité. Une entité peut être une personne, un lieu ou un objet considéré comme nécessaire pour l'interaction entre un utilisateur et une application, incluant l'utilisateur et l'application.

Dans l'article [78], les auteurs comparent six méthodes de modélisation de contexte dont la modélisation orientée objets et la modélisation basée sur les ontologies. La modélisation orientée objets (utilisé par le projet EnTiMid) met à profit les aspects héritage et réutilisation des propriétés des objets modélisés. Mais, d'après les auteurs, les meilleurs types de modélisations sont ceux basés sur les ontologies (par exemple Dog). Les ontologies fournissent un meilleur niveau de formalisation. L'utilisation de cette technologie permet de produire de nouvelles connaissances avec des éléments de connaissance de base.

1.2 Aspects techniques

Avant de décrire quelques projets en relation avec la domotique, cette section permet de donner des informations sur les technologies citées dans la suite de ce mémoire :

- *UniversalPlug and Play* (UPnP) est une infrastructure de communication pour les dispositifs ayant moins de contraintes que les dispositifs utilisant Zigbee.
- ZibBee quant à lui, est une infrastructure de communication destinée aux dispositifs contraints en mémoire et en capacité de calcul.
- *Open Service Gateway initiative* (OSGi) est une plate-forme d'exécution d'applications java.

1.2.1 UPnP

UPnP est un ensemble de protocoles de configuration de systèmes et de découverte de services conçus pour les ordinateurs et les dispositifs communicants utilisant les couches TCP, UDP et DHCP. Ces protocoles permettent, au sein d'un écosystème UPnP d'inclure un nouvel objet communicant, de découvrir les objets communicants et ses services associés, de présenter les services, et enfin d'utiliser les services. La méthode d'utilisation des services est donnée par la description des services. La présentation des services permet à un utilisateur de recevoir des informations sur les dispositifs au travers une page XML, et l'utilisation d'un service se fait par la demande d'une ressource HTTP.

Le protocole d'autoconfiguration d'adresse DHCP permet d'attribuer une adresse IP à un nouvel objet, ce dernier peut ainsi s'intégrer dans l'écosystème UPnP en diffusant son identité et ses services. La diffusion et les demandes d'informations relatives aux services et aux objets communicants présents sont supportées par un protocole de communication basé sur UDP, le Simple Service Discovery Protocol (SSDP). Le protocole SSDP utilise des messages SOAP au format XML ; ces derniers sont diffusés sur le protocole HTTPU (HTTP sur un canal UDP) en multicast.

L'utilisation de cet ensemble de protocoles est consommateur en mémoire, en bande passante et en ressource de calcul, ces deux derniers éléments ayant un effet notable sur l'énergie consommée par les objets communicants. Cet ensemble de protocoles n'est pas compatible avec les attentes des dispositifs embarqués, notamment les dispositifs de la maison. Les fabricants des objets de la maison cherchent des technologies pour réduire la consommation d'énergie, l'utilisation de la mémoire et la bande passante. Une solution pour ces préoccupations est par exemple l'écosystème Zigbee.

1.2.2 Zigbee

La technologie Zigbee est utilisée pour échanger des données grâce à des médias de communication sans fil. Zigbee est conçu pour la supervision et le contrôle d'un environnement composé de dispositifs mis en réseau et disposants de batteries. Il prend en compte la faible puissance électrique et de calcul des dispositifs ciblés et permet aussi de sécuriser les transmissions de données. Ce protocole de communication est un standard ouvert, les spécifications pouvant être téléchargés sur le site de Zigbee Alliance [87].

Zigbee fournit différents niveaux d'interopérabilité entre les dispositifs. Pour assurer la compatibilité des communications des dispositifs qui implémentent le protocole, Zigbee Alliance impose une série de tests des micrologiciels. Les micrologiciels passant les tests reçoivent la certification Zigbee Compliant Platform (ZCP). En plus de la compatibilité des micrologiciels, le protocole propose une couche applicative décrivant des objets client ou serveur. Cette couche applicative définit des profils de dispositifs et, là encore, Zigbee Alliance impose des tests pour estampiller les dispositifs de la compatibilité avec un profil.

Les caractéristiques de Zigbee sont issues de quarante années d'expérience dans le domaine de la communication radio de l'institut IEEE. Les spécifications de Zigbee sont décrites dans différents documents. L'infrastructure Zigbee a été créée pour le contrôle et la surveillance d'environnements, mais la couche applicative n'est pas figée. Ainsi, il est possible de faire passer de la voix ou d'établir un tunnel de données. Afin de créer une compatibilité entre les équipements d'un même domaine, Zigbee propose des profils pour lesquels l'interopérabilité est assurée par la spécification de la couche applicative. Les profils d'application spécifiés et ouverts sont par exemple : Home-Automation pour la domotique, Smart Energy pour la gestion de la consommation énergétique et RF4CE (*Remote Control Standard for Consumer Electronics*) pour les télécommandes.

Zigbee est conçu pour répondre aux besoins de fiabilité des communications [68], de sécurité (cryptage AES-128 bit), de faible consommation et de faible coût par rapport aux coûts des dispositifs dans lequel Zigbee est implémenté. Pour cela, Zigbee utilise une faible vitesse de communication. Lorsque les dispositifs sont dans un environnement non perturbé, les communications se font à une vitesse de 250 kilo-octets par secondes. Ces échanges se font grâce à des trames ne dépassant pas 128 octets.

Pour assurer la fiabilité de la transmission, Zigbee utilise différentes technologies comme : deux modulations de fréquence pour s'adapter aux bruits ambiants, la règle du CSMA-CA, des corrections de données sur plusieurs couches OSI, etc. La topologie du réseau Zigbee permet d'augmenter la fiabilité de la transmission, notamment grâce à un algorithme élaboré de routage [61].

1.2.3 OSGi

Java est un langage ne proposant aucun moyen de créer des modules. Les archives Java (fichiers jar) contenant une partie des fichiers nécessaires à une application ne sont pas des modules. En effet, une fois la machine virtuelle Java chargée, l'information qu'un fichier appartient à un jar ou un autre n'est plus disponible. Ce qui a pour conséquence que toutes les ressources publiques sont disponibles sans aucune restriction. Qu'arrive-t-il lorsqu'une application a besoin de deux versions d'une ressource ? Le langage Java n'apporte pas de réponse, il faut utiliser des techniques spéciales pour contourner ce problème. Mais, ces techniques détournent le programmeur de sa principale préoccupation.

OSGi est une spécification d'un framework de gestion de la modularité d'une application Java. Cette plate-forme permet de contrôler dynamiquement le cycle de vie de composants logiciels s'exécutant sur une machine virtuelle Java. Le framework fourni pour développer une application impose son découpage sous forme de composants. Lors de l'exécution, la plate-forme OSGi s'occupe de résoudre les dépendances entre les composants et gèrent les différentes versions des composants chargés en mémoire. De plus, les plate-formes OSGi fournissent un ensemble d'interfaces de programmation (*Application Programming Interface* ou API) pour les tâches communes comme la journalisation ou les préférences.

1.3 Pont entre deux protocoles de communication

Les ponts sont des éléments de traduction entre deux protocoles de communication. Ils permettent de transoporter les messages de l'un vers l'autre. Les ponts permettent ainsi l'interopérabilité entre les dispositifs, agissant en tant que médiateur permettant l'interaction entre les éléments des différents écosystèmes.

1.3.1 Pont UPnP-Zigbee

L'article [57] présente un pont UPnP-Zigbee. Il a pour but la promotion de l'infrastructure UPnP en présentant des techniques de traduction de protocole d'objets communicants vers UPnP. Le pont UPnP-Zigbee présenté dans cet article a comme objectif de commander des dispositifs Zigbee depuis une infrastructure réseau UPnP. UPnP est ainsi utilisé en tant qu'élément pivot et permet d'exposer les dispositifs Zigbee sur un environnement réseau aisément manipulable par des outils informatiques.

Le pont UPnP-Zigbee emploie deux technologies de communication qui ont chacune des préoccupations opposées. Les différences techniques induites par ces préoccupations doivent être converties d'une technologie à l'autre afin de créer

l'interopérabilité. La traduction entre l'architecture Zigbee et l'écosystème UPnP est réalisée par la mise en place de composants logiciels. Ces composants ont pour but de virtualiser les dispositifs Zigbee dans l'écosystème UPnP en tant que services. Ainsi, les éléments de contrôle de l'écosystème UPnP peuvent interagir avec les dispositifs Zigbee et les éléments d'information Zigbee (capteurs ou éléments de contrôle) peuvent donner des informations aux éléments UPnP. En résumé, le pont UPnP-Zigbee injecte des dispositifs virtuels dans une infrastructure UPnP.

1.3.2 Amigo

Le projet IST Amigo (*ambient intelligence for the networked home environment* disponible à l'adresse http ://www.hitech-projects.com/euprojects/amigo/) vise à faciliter la mise en relation spontanée d'objets communicants pour offrir des services de haut niveau aux utilisateurs tout en prenant en compte le contexte d'utilisation. L'approche du projet Amigo est basée sur la définition d'un paradigme standard pour l'accès aux dispositifs et le développement de composants spécialisés de traduction des protocoles de communication (composant *proxy*) afin d'adapter les technologies existantes.

Ce projet utilise le paradigme d'architecture orientée services afin d'offrir une solution domotique qui s'adapte aux demandes des utilisateurs. Amigo peut-être divisé en deux parties : une partie bas niveau [28] et une partie intelligence. La partie bas niveau offre un ensemble de fonctionnalités qui facilitent l'interopérabilité entre les services comme la découverte de services et l'interaction entre services. La partie intelligence contient les fonctionnalités qui permettent de faciliter la création de la solution adaptée à l'utilisateur. Ces fonctionnalités sont disponibles sous forme de services.

Amigo n'impose pas de protocoles particuliers pour l'interopérabilité [29], mais leur utilisation conjointe doit offrir la possibilité de faire la découverte de services, d'échanger des messages et d'envoyer des évènements. UPnP et les service web (*Web Service* ou WS) sont des exemples de protocoles compatibles avec les services Amigo. La méthode utilisée pour l'intégration des objets communicants est intéressante. Les objets communicants sont introduits dans la solution Amigo par l'intermédiaire de composants logiciel *proxy*. Les ponts sont inclus dans la partie bas niveau de Amigo. Ces ponts transforment les fonctionnalités et les notifications des objets communicants en services et évènements mis à disposition sur les protocoles choisis (UPnP, WS ou autre).

Les dispositifs ainsi que les applications dans un environnement Amigo sont abstraits par des services. L'interopérabilité entre les services est matérialisée par l'utilisation de descriptions sémantiques. Plus précisément, chaque service Amigo

possède la faculté de décrire formellement ses fonctionnalités au moyen d'un ensemble de termes représentant un domaine spécifique. Ce type de description est interprétable par un élément algorithmique afin de comprendre les fonctionnalités offertes par un service. Cette méthode de description est présentée dans le chapitre suivant. Elle donne l'opportunité de faire évoluer les termes de description des fonctionnalités et permet ainsi aux composants Amigo de gérer, dans le futur, la description de nouvelle fonctionnalités.

La définition de contexte dans le cadre du projet Amigo [50] est plus générale que de la définition donnée dans la section précédente. En plus des données physiques de l'environnement, le contexte prend en compte les différents services disponibles dans l'environnement, par exemple les interfaces graphiques. La gestion du contexte est réalisée par un composant logiciel utilisé par les applications incluses dans un environnement Amigo. La représentation du contexte et le contexte lui-même sont décrits par une ontologie.

1.4 Modèle extensionnel

1.4.1 EnTiMid

Le projet EnTiMid propose un retour d'expérience sur la construction d'une architecture logicielle pour un système domotique mutliprotocole. Le but d'EnTiMid est de réaliser différents types d'accès à des protocoles de communication d'objets communicants. Les principaux objectifs du projet sont exprimés sous forme d'exigences dans l'article [64]. Ces objectifs sont : l'interconnexion des objets communicants, l'ouverture à des protocoles de communication haut niveau (UPnP, *Simple Object Access Protocol* (SOAP)...), la gestion de l'évolution dynamique du système, de la sécurité et de la sûreté de fonctionnement, et de favoriser les accès nomades.

EnTiMid est un intergiciel composé de trois couches d'abstraction. La couche basse permet la substitution des protocoles domotiques, c'est une couche de traduction des différents langages d'équipements vers le niveau intermédiaire. Le niveau intermédiaire permet une représentation abstraite des dispositifs domotiques et permet aussi d'introduire l'interopérabilité entre les équipements. Un troisième niveau transforme la représentation abstraite en services accessibles depuis l'extérieur de l'intergiciel EnTiMid grâce à des technologies comme SOAP ou UPnP [49, 64].

EnTiMid est développé en Java pour une plate-forme OSGi [65]. Le fait que l'application utilise une plate-forme OSGi permet d'offrir trois caractéristiques intéressantes :

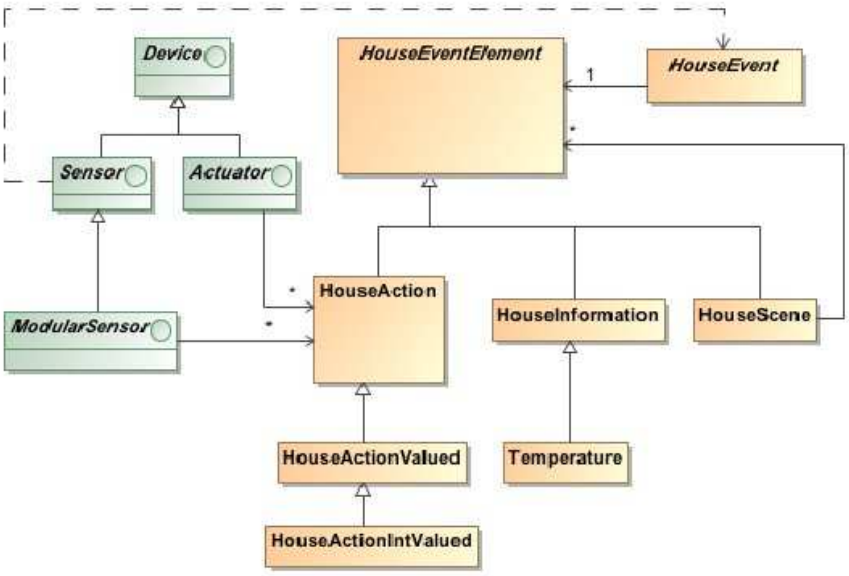


FIGURE 1.1 – Modèle de la couche d'abstraction d'EnTiMid

- La dynamicité permet de démarrer où d'arrêter certains services, de changer les paramètre de l'application en cours d'exécution.
- L'évolutivité permet d'ajouter de nouveaux protocoles sans modifier les développements précédents.
- Enfin, la multipersonnalité permet la communication entre des technologies différentes.

La figure 1.1 présente le modèle UML de la couche d'abstraction des protocoles d'EnTiMid. Pour qu'EnTiMid communique avec un nouveau protocole, il faut implémenter un module traducteur conforme au modèle générique d'EnTiMid exprimé sous forme d'un ensemble d'interfaces et de classes génériques. C'est à travers le portage des différents moyens d'accès aux technologies que l'utilisation transparente des traducteurs est atteinte. Le modèle de données représentant la couche d'abstraction décrit les différents capteurs-actionneurs permettant de déclencher ou écouter les scénarios préprogrammés (les *HouseActions*).

L'évolutivité d'EnTiMid est un point important de ce projet. EnTiMid est réalisé grâce à une architecture en couche orientée composants. Cette architecture à base de composants permet de réutiliser certaines parties du code pour d'autres modules et ainsi faciliter l'intégration de nouveaux protocoles. L'adaptation dynamique dans EnTiMid est réalisée via la plate-forme OSGi sous-jacente.

1.4.2 Pise et SStreaMWare

Le but de PISE [7] est de fournir une infrastructure logicielle pour gérer des services de haut niveau dans le domaine de la distribution électrique. Ce projet propose une architecture hiérarchique qui utilise un grand nombre de plates-formes intermédiaires fournissant un ensemble de services aux clients finaux. PISE a pour objectif de proposer :

- un ensemble de services non fonctionnels afin d'aider à la réalisation de services métiers à forte valeur ajoutée,
- la capacité d'embarquer des services tiers provenant de fournisseurs tiers,
- une administration distante d'un grand parc d'équipements et des services qu'ils hébergent.

PISE est un ensemble d'outils pour générer, déployer et superviser les applications et les services fonctionnant sur des passerelles Internet distantes. Les services fonctionnant sur les passerelles sont capables de s'interconnecter en utilisant différentes infrastructures de communication (par exemple UPnP, CORBA, JINI, ect.).

Dans la même lignée, SStreaMWare [42, 41] est un intergiciel qui adopte une approche orientée services. Les fonctionnalités de récupération des données auprès des capteurs sont fournies en termes de services afin de cacher l'hétérogénéité de ceux-ci. Ces services implémentent des modèles de données et de requêtes, qui permettent d'évaluer des requêtes complexes sur les flux de données issues de capteurs. L'arrivée, le départ, la modification des capteurs et des services sont pris en compte de manière dynamique dans SStreaMWare.

Les mécanismes de requêtes élaborées sur des données, disponibles dans SStreaMWare, ou sur les évènements d'une application permettent de décrire des scénarios.

1.5 Modèle conceptuel

1.5.1 Dog

Le but de Dog [9] (Domotic OSGi Gateway) est de fournir une application permettant de créer une passerelle domotique intelligente multiprotocole commandant les objets communicants de la maison. Ce projet doit servir de base à l'implantation d'une "maison intelligente". Pour cela, Dog utilise DogOnt [10], un modèle de la maison basé sur les ontologies [35]. DogOnt permet de décrire les objets communicants et l'architecture d'une maison indépendamment des technologies utilisées. Les états et les fonctionnalités sont automatiquement associés aux éléments d'une installation domotique grâce à l'utilisation des mécanismes d'interrogation des ontologies.

Dog utilise une plate-forme OSGi afin de gérer le déploiement de l'application et le cycle de vie des composants de Dog. Dog peut-être divisé en quatre couches :

- La première couche contient la gestion des composants Dog ainsi que les services non fonctionnels de l'application.
- La seconde couche fournit des composants logiciels d'abstraction des écosystèmes des objets communicants de la maison au moyen d'un modèle de représentation commun basé sur une ontologie.
- La troisième couche fournit un composant de médiation des messages de l'application. Ce dernier redistribue les messages de commande aux composants de la couche inférieure en fonction de la destination du message et crée un écosystème virtuel où tous les objets communicants sont accessibles. De plus, cette couche héberge les composants d'intelligence.
- Enfin la quatrième couche permet d'interfacer Dog avec des applications extérieures au moyen d'APIs et de communications de type client-serveur.

L'application Dog propose une architecture s'occupant des préoccupations liées à l'accès aux objets communicants de la maison et à la modélisation de ces objets. La quatrième couche permet de rendre l'application Dog modulaire. Elle permet l'accès à ses fonctionnalité par l'intermédiaire d'un ensemble de service web et d'un ensemble d'API. La préoccupation liée à l'utilisation et la présentation des fonctionnalités est repoussée vers des applications externes. Par exemple, le site du projet [19] propose une interface graphique de commande d'une installation fictive réalisée en C# ainsi qu'une application de suivi de la température d'un capteur réalisé en Java.

Au sein de l'écosystème Dog, les objets communicants sont abstraits par un modèle formel de la maison défini par l'ontologie DogOnt. En plus de modéliser les objets communicants, un des objectifs de l'ontologie permet de prendre en compte le contexte, c'est-à-dire l'état d'une installation domotique. En effet, DogOnt modélise à la fois l'état des dispositifs et leurs fonctionnalités associées. L'article [11] montre comment un composant d'intelligence Dog utilise le contexte d'une installation domotique pour raisonner et créer de nouvelles données de contexte.

1.5.2 Oasis

Le but d'Oasis (*Open architecture for Accessible Services Integration and Standardization*) est de fournir un ensemble de services qui permettent d'améliorer la vie des personnes âgées [66]. Oasis s'occupe de fournir une solution technologique à quatre préoccupations qu'ont les personnes âgées : socialisation, automatisation de la maison, aide à l'autonomie et aide à la mobilité. Pour cela, les objectifs sont

d'introduire une architecture supportée par des ontologies et de développer un système composé d'applications pouvant interagir ensemble.

Pour créer l'interopérabilité et le partage d'informations du contexte entre les applications et les objets communicants, il faut extraire les fonctionnalités de chacun d'entre eux et fournir une description de leur utilisation. Les participants du projet Oasis font le constat que l'ajout de nouvelles fonctionnalités décrites de manière *ad hoc* introduit des effets de bords lors de leur intégration. La solution qu'apporte Oasis est de modéliser les services dans l'ontologie [55]. Les auteurs de cet article introduisent le terme de *service sémantique*.

La modification de l'ontologie est une opération délicate, en particulier lorsqu'elle est modifiée par différentes personnes. Pour cela, les auteurs des rapports [54, 6] introduisent l'utilisation d'un outil basé sur un mécanisme de type Wikipédia afin de :

- permettre un travail collaboratif pour l'évolution de l'ontologie,
- faciliter l'intégration des différentes parties de l'ontologie.

L'aspect travail en équipe sur les ontologies est un point important abordé par le projet Oasis. Dans ce projet, les applications développées doivent fournir une définition sémantique de leurs fonctionnalités. Ce qui implique que les ontologies d'applications doivent se référer à une ontologie de plus haut niveau et ne pas définir des concepts déjà définis par une autre application. La solution apportée par Oasis est de décrire une ontologie centrale nommée hyper-ontologie [6] où les ontologies d'applications se « connectent » tout en restant indépendantes les unes des autres. La connexion des ontologies d'applications avec l'hyper-ontologie est réalisée par un framework.

1.5.3 PERSONA

Le but de Persona [80] (Perceptive Spaces prOmoting iNdepentent Aging) est de spécifier une architecture logicielle pour le maintien des personnes à la maison. L'objectif est de fournir une plate-forme évolutive et adaptable aux besoins des utilisateurs. Persona est un ensemble de composants (l'article indique modules en lieu et place de composants) disposés autour d'un intergiciel fondé sur une plate-forme OSGi. L'intergiciel est construit au moyen d'une architecture repartie basée sur des bus de communication. Les composants de Persona sont mise en place sur le modèle d'une architecture basée sur les services. L'interopérabilité entre les services est réalisée par l'utilisation de descriptions sémantiques. Cette technologie est inspirée du projet Amigo présenté dans la sous-section 1.3.2 de ce mémoire.

Les composants de Persona fournissent des services qui permettent de composer une application adaptée aux besoins des utilisateurs. Cependant, la base d'une ap-

plication Persona repose sur un ensemble de fonctionnalités indispensables. Chacune des fonctionnalités est fournie par un composant. Ces fonctionnalités sont :

- L'historique du contexte qui permet aux autres composants d'interroger l'état courant et passé de l'installation. Ce composant est nécessaire, car la modification de l'état d'un dispositif ou d'un service est envoyée par un évènement et il n'est pas toujours possible d'interroger le composant d'origine.
- Un raisonneur de contexte qui permet d'inférer des données suivant le contexte.
- Une passerelle logicielle qui interface différents types d'entrées/sorties comme l'avertissement par SMS ou le pilotage par téléphone.
- Un orchestrateur de service qui permet de gérer les compositions de services.
- Une interface utilisateur qui permet de fournir automatiquement aux usagers une vue des services disponibles.

Les composants collaborent entre eux au moyen d'un intergiciel fournissant des bus de messages. Il existe quatre bus de messages, ce qui permet de créer un faible couplage entre les composants et donne la possibilité aux composants de collaborer. Ces quatre bus de messages sont :

- le bus de contexte utilisé pour avertir un changement dans le contexte,
- le bus de service qui fournit un moyen de chercher les services disponibles dans l'application,
- un bus de sortie qui choisit le meilleur composant afin d'informer et d'interroger les usagers de l'application,
- un bus d'entrée qui permet l'entrée de données depuis le monde réel vers l'application.

Lors qu'un composant démarre, il s'enregistre sur les différents bus en tant qu'émetteur ou receveur puis s'annonce sur le bus de service. Les bus ont pour rôle d'acheminer les messages aux composants. La détermination des destinataires d'un message est réalisée grâce à une analyse sémantique des besoins des composants. Cette technologie est reprise du projet Sodapop [23].

1.6 Synthèse

Dans ce chapitre, nous avons décrit sept projets différents de par leur préoccupation principale. Ces préoccupations ont abouti à des choix techniques que nous

avons analysés au moyen de huit critères présentés en section 1.1. Le tableau 1.1 rappelle les choix de six des solutions en fonction de ces critères. Lorsque le contenu d'une cellule est *NA*, cela signifie que les papiers cités dans la section n'abordent pas ce critère.

Dans l'article [78], les auteurs indiquent que les ontologies et l'approche objet sont les meilleurs techniques pour modéliser le contexte avec une préférence pour les ontologies. Remarquons que dans les projets présentés, les ontologies sont souvent utilisées lorsqu'il s'agit de modéliser le contexte. Nous pouvons aussi remarquer que lorsqu'un outil aussi flexible que les ontologies est utilisé, l'abstraction du matériel est aussi supportée par les ontologies. Le chapitre suivant décrit le mécanisme des ontologies. Suivant la définition du contexte de la section 1.1.4 c'est évident : le contexte doit comprendre les descriptions de tout ce qui est nécessaire pour satisfaire les objectifs d'une application. Donc, lorsque des dispositifs matériels sont utilisés par une application, leurs fonctionnalités doivent être modélisées dans le contexte et donc l'ontologie.

Oasis est Persona ne présentent pas d'abstraction du matériel, car ce n'est pas la principale préoccupation de ces projets. En effet, leur préoccupation est de fournir une plate-forme d'environnement ambiant. De ce fait, ces deux projets n'abordent pas directement la problématique d'abstraction du matériel, mais comment offrir des services compréhensibles par les autres acteurs du système.

Ces projets s'efforcent de trouver des solutions d'architectures les plus génériques possible pour héberger des applications d'environnement intelligent, mais aucun ne parle du comportement d'une installation domotique.

Critères		Pont Zigbee-UPnP	Amigo	Entimid	Dog	Oasis	Persona
Modélisation conceptuelle		Composants	Ontologie	Modèle orienté objets	Ontologie	Ontologie	Ontologie
Abstraction du matériel		Services UPnP	Ontologie	Figé par le modèle	Ontologie	NA	NA
Intergiciel	Repartition	Pont	WS	OSGi	OSGi et RPC	SOAP	OSGi
	Compatibilité	Zigbee et UPnP	À définir à priori (SOAP ou autres)	Java	Java et programmes implémentant XML-RPC	Programmes implémentant SOAP	Java
	Modularité	Services UPnP	WS	Communication par messages	DogMessage	SOAP et service sémantique	Quatre types de messages
	Gestion de composants	NA	platforme OSGi	OSGi	OSGi	NA	OSGi
	Gestion du système	NA	Composant ad-hoc	NA	NA	NA	NA
Prise en compte du contexte		NA	Composant ad-hoc basé sur une ontologie	NA	Ontologie	Ontologie	Composant *ad-hoc* basé sur une ontologie

TABLE 1.1 – Résumé des caractéristiques des projets

Sommaire

Chapitre 2

Les Ontologies

Les ontologies sont une formalisation de la représentation et de la modélisation de la connaissance. La connaissance représentée par une ontologie est un ensemble d'objets, de concepts et d'entités d'un domaine et de leurs relations, elle ne possède un sens que pour ceux qui peuvent l'interpréter. Les ontologies peuvent être utilisées pour les communications entre acteurs, les acteurs étant des personnes ou des entités algorithmiques. Ce type d'ontologies permet de définir les concepts généraux, de donner un nom à ces concepts et de définir les relations entre les concepts. Les ontologies peuvent aussi être utilisées pour leur cadre formel et contraint. Dans ce cas, des éléments algorithmiques peuvent raisonner sur la connaissance contenue dans une ontologie, inférer de nouvelles connaissances et prendre une décision. Les ontologies peuvent être utilisées pour leur aspect relations entre concepts, ce qui permet aux applications logicielles de « voir » les concepts de différents points de vue. Enfin, les ontologies peuvent être utilisées lors de la phase de développement d'une application.

Les ontologies modélisent des concepts à différents niveaux d'abstractions. Dans la littérature, N. Guarino [36] différencie trois niveaux d'abstraction représentés par la figure 2.1 :

- Les ontologies de haut niveau décrivent des concepts très généraux indépendamment d'un domaine ou d'un problème particulier.
- Les ontologies de domaines et les ontologies de tâches sont une spécialisation des ontologies de haut niveau. Pour cela, elles spécialisent les

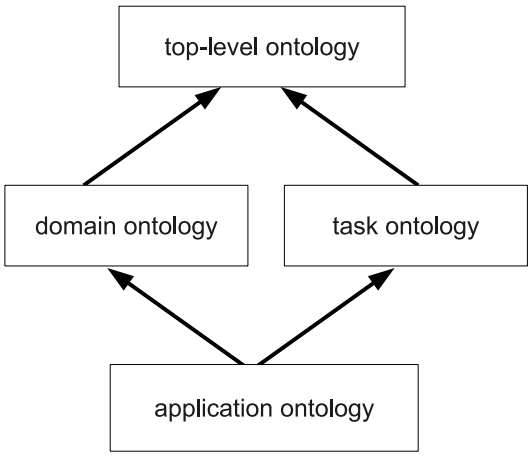


FIGURE 2.1 – Type d'ontologies selon Guarino [36].

termes introduits dans l'ontologie de haut niveau. Une ontologie de domaine décrit le vocabulaire lié à un domaine (exemple : la médecine, les automobiles, etc.). Une ontologie de tâches décrit les tâches ou les activités d'un domaine (exemple : diagnostic médical, vente de voiture, etc.).

– Les ontologies d'applications décrivent des concepts dérivés des deux types d'ontologies du niveau précédent. Ces ontologies représentent le niveau de modélisation le plus proche du problème à résoudre. Les concepts modélisés peuvent utiliser un système d'héritage multiple pour spécialiser les concepts plus génériques.

2.1 Origine et définitions contemporaines

À l'origine, le terme Ontologie est utilisé pour définir le domaine de la philosophie qui étudie « l'être en tant qu'être ». Dans ce cadre, l'Ontologie cherche à définir les différents modes d'existence des objets et leurs propriétés ; les objets sont des entités qui sont perçues de manière identique par de nombreuses personnes, par exemple : une matrice, la langue française, l'âme... Plus de détails sur l'historique et la définition philosophique de l'Ontologie sont disponibles sur le site internet [74].

Dans les années 1970, la discipline de l'intelligence artificielle (IA) est confrontée au problème de représentation et d'exploitation de la connaissance. La communauté de cette dernière met au point une formalisation d'un modèle de la connaissance et des outils qui permettent d'utiliser cette connaissance. Dans les années 1980, cette communauté commence à utiliser le terme « ontologie » pour définir à la fois la formalisation du modèle et le composant qui stocke la connais-

sance. À partir des années 1990, d'autres disciplines comme la recherche médicale ou le WEB sémantique commencent à utiliser les ontologies. Elles appellent ce domaine l'ingénierie des connaissances. L'ingénierie des connaissances est l'étude des concepts, méthodes et techniques permettant de modéliser et d'acquérir les connaissances pour les systèmes réalisant ou aidant des humains à réaliser des tâches. Durant ces années, un effort important est produit pour créer des ontologies de haut niveau (cf. top-level ontology figure 2.1) comme la base de connaissances Cyc [1] ou l'ontologie Wordnet [2] dont les projets débutèrent respectivement en 1984 et en 1985.

Depuis les années 1990, les chercheurs proposent différentes nuances et définitions du terme ontologie. La plus générale et la plus connue est la définition de Thomas R. Gruber de 1993 [34] où il décrit qu'une ontologie est une spécification explicite d'une conceptualisation d'un domaine de connaissance :

> « An ontology is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an Ontology is a systematic account of Existence. For artificial intelligence systems, what exists is that which can be represented. »

Cette définition est élégante puisqu'elle utilise des termes techniques et précis relatifs à l'informatique. De plus, T.R. Gruber confirma cette définition en 2009 dans l'article [35].

En 1995, dans [37] les auteurs introduisent la notion d'ontologie formelle, notion reprise par l'article [82] pour classer les ontologies en différents niveaux de formalisation. Ce dernier article introduit la notion d'ontologies implicites, peu utilisée, qui correspond à la connaissance d'un domaine, mais qui ne peut pas être partagée.

En 1995, C. Roche [69] donne une définition du terme « ontologie » tel que nous l'utilisons dans la suite de ce mémoire :

> « Définie pour un objectif donné et un domaine particulier, une ontologie est, pour l'ingénierie des connaissances, une représentation d'une modélisation d'un domaine partagée par une communauté d'acteurs. Objet informatique défini à l'aide d'un formalisme de représentation, elle se compose principalement d'un ensemble de concepts définis en compréhension, de relation et de propriétés logiques. »

Pour conclure, l'ontologie peut être interprétée sous différentes formes. Une forme sémantique comme le définit T. R. Gruber, où l'ontologie est un moyen de spécifier un vocabulaire en contraignant la modélisation du modèle ou une forme

1. http ://www.opencyc.org/
2. http ://wordnet.princeton.edu/

syntaxique comme le définie C. Roche, où l'ontologie est un modèle mathématique proche de la logique. Dans ce mémoire, nous utilisons le terme ontologie comme une ontologie formelle explicite utilisée en tant qu'élément informatique ayant une finalité applicative. La section suivante présente ce que sont les concepts, les relations et les propriétés logiques.

2.2 Notions liées aux ontologies

Il convient de rappeler qu'une ontologie permet de représenter la vision d'une partie du monde relative à un domaine et à une application logicielle. Cette vision du monde est composée d'un ensemble de concepts, d'instances, de définitions, de relations, de propriétés et d'une hiérarchie entre les concepts. Un concept représente une idée et est définie :

- en intension (ou en compréhension) au moyen de ses relations, ses propriétés et sa hiérarchie ;
- en extension au moyen des instances liées au concept ;
- ou par plusieurs symboles.

Notons que le lien entre les instances et les concepts ainsi que la hiérarchie sont des relations particulières.

2.2.1 Concepts et instances

Les concepts sont exprimés, en général, par un ou plusieurs termes ou symboles. Un concept représente une classe d'instance, l'ensemble de ses instances est l'extension du concept. D'un point de vue ensembliste, les instances sont disjointes des concepts, et leur union est l'ensemble des objets de l'ontologie. L'intension d'un concept est l'ensemble des attributs, propriétés et relations, c'est la sémantique du concept. Autrement dit, l'intension d'un concept est le sens tel qu'il est interprété par les éléments algorithmiques. La figure 2.2 illustre un exemple du concept d'une entrée définie en intension et en extension lié à plusieurs termes et un symbole. Cette figure présente le concept « d'entrée », mais ne présente pas les concepts qui lui sont associés comme l'ouverture et le mur.

L'ontologie est un élément informatique qui représente un modèle d'un domaine. La cohérence du domaine modélisé par une ontologie est vérifiée au moyen de règles définies par le formalisme et par des règles métiers :

- Les règles métiers sont des contraintes d'intégrité définies lors de la création de l'ontologie. Ces règles sont définies sur les relations entre les concepts.
- Les règles définies par le formalisme des ontologies sont des propriétés des concepts.

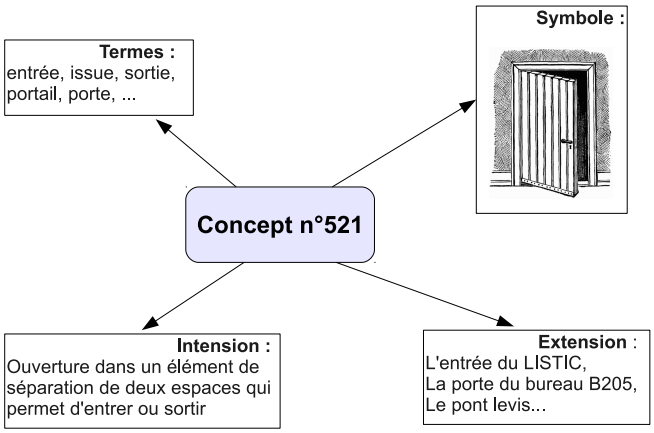


FIGURE 2.2 – Exemple du concept d'une entrée.

Dans [38] les auteurs listent les propriétés de bases associées aux concepts des ontologies. Les propriétés portant sur un concept sont : la rigidité, l'unité et l'identité.

Un concept est rigide si toutes ses instances restent instance de ce concept. Du point de vue extensionnel, un concept rigide est essentiel pour caractériser l'instance. Inversement, un concept est antirigide si celui-ci n'est pas essentiel à une instance. Par exemple, *humain* est un concept rigide alors *étudiant* est un concept antirigide. La propriété d'identité donne l'information sur la capacité d'un concept à permettre de différencier ses instances. Par exemple, le numéro d'étudiant permet d'identifier un étudiant. La propriété d'unité permet de construire un élément à partir de plusieurs parties. Dans le domaine de la médecine, un humain est composé de membres et d'organes par exemple.

2.2.2 Relations entre les concepts et les réalisations

La connaissance d'une ontologie est représentée par ses concepts et par les relations tissées entre ses concepts. Ces relations sont exprimées, en général, par un terme ou un symbole. Les réalisations sont l'extension des relations à l'instar des instances qui sont l'extension des concepts. L'intension d'une relation est l'ensemble des attributs et propriétés communes à toutes les réalisations d'une relation, c'est la sémantique de la relation. De plus, des contraintes peuvent être attribuées au domaine et au codomaine des relations, c'est la signature de la relation. Une relation peut être considérée comme une fonction binaire qui représente une association d'un concept source à un concept destination. Les relations sont divisées en deux types : les relations hiérarchiques et les relations associatives.

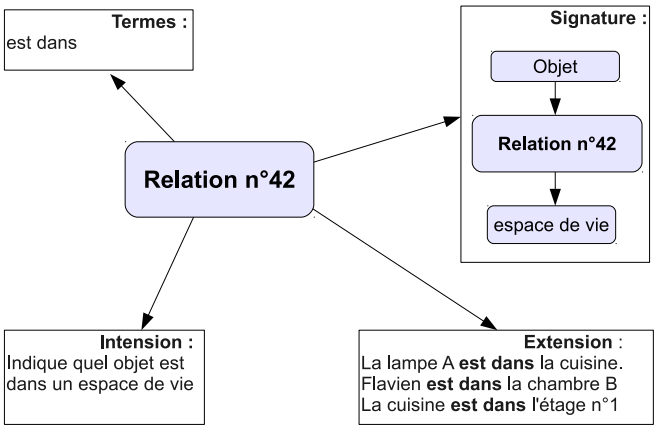


FIGURE 2.3 – Exemple de la relation entre un objet et un lieu de vie.

Les relations hiérarchiques permettent de classer les concepts entre eux. Ainsi, des concepts peuvent être équivalents dans le cas de synonymes, disjoints si l'extension des concepts est incompatible, sous-concept d'un concept et être une instance d'un autre concept. La relation « A est sous-concept de B » permet au concept A d'hériter des propriétés de B.

Le concept A doit se différencier de B en définissant de nouvelles propriétés, sinon A doit être défini en tant que concept équivalent de B.

La relation de concepts disjoints correspond à la notion ensembliste portant sur les instances des concepts. Par exemple si `Humain` est disjoint de `Animal`, et que `Patrice` est une instance de `Humain`, alors `Patrice` ne peut pas être une instance de `Animal`. La relation « est une instance de » permet de définir les extensions d'un concept.

Les relations associatives sont définies lors de la mise au point de l'ontologie. Elles permettent de définir les relations entre les concepts qui existent dans le domaine. La figure 2.3 illustre un exemple de relation associative entre un objet et un lieu de vie. La signature de la relation impose que l'objet destination de la relation soit un lieu de vie. Cette relation peut être considérée comme la fonction : $est\_dans(flavien, chambreB) = true$.

À l'instar des concepts, les relations disposent aussi de propriétés qui permettent de vérifier la connaissance de l'ontologie. Les relations peuvent être symétriques, transitives, réflexives, fonctionnelles :

- Une relation est symétrique si elle peut-être appliqué dans les deux sens ;
- Une relation est transitive lorsqu'un objet intermédiaire permet de définir une relation entre deux objets. Par exemple si $\mathcal{R}$ est transitive et $\mathcal{R}(A, B) \wedge \mathcal{R}(B, C)$, nous pouvons déduire que $\mathcal{R}(A, C)$;

– Une relation est réflexive si un objet peut-être associé à lui même : $\mathcal{R}(A, A)$;

– Et enfin, une relation est fonctionnelle si elle définit pour un objet au plus un objet unique (par exemple l'âge).

2.2.3 Inférence ou raisonnement

Une ontologie est une entité représentant des connaissances de manière formelle. Le caractère formel permet aux ontologies d'être traitées par des moteurs d'inférences. Ce terme est utilisé pour indiquer un élément informatique qui permet de vérifier les connaissances de l'ontologie et aussi de trouver les connaissances implicites du domaine. Les connaissances implicites sont déduites au moyen des relations entre les concepts et de règles définies. L'inférence est le processus de création de connaissance à partir d'un raisonnement de déduction ou d'induction. Le phénomène d'inférence appliqué à un humain pourrait être : lorsqu'une lumière d'une maison s'allume, un passant peut déduire que cette maison est habitée. Les moteurs d'inférence permettent aussi de vérifier les connaissances en utilisant les contraintes définies sur les règles et les concepts.

Les règles permettent de décrire des relations métiers qui ne peuvent pas être modélisées par les relations. Ces règles sont exprimées au moyen d'un formalisme proche de la logique du premier ordre, impliquant les quantificateurs. Elles peuvent être incluses dans l'ontologie, dans ce cas elles modélisent la connaissance opérationnelle du domaine et le moteur de raisonnement les prend en compte afin de créer un nouvel élément de connaissance. Elles peuvent aussi être utilisées en cours d'exécution et dans ce cas elles se comportent comme des requêtes.

Par exemple, la règle $Pere(X, Y) \wedge Homme(Y) \implies Fils(Y, X)$ détermine qu'un enfant Y est fils d'une personne X si Y est un homme et si X est son père. Cette règle permet de conclure automatiquement la relation fils à partir des postulats Père et Homme.

2.3 Utilisations

Selon l'article [82], les ontologies sont utilisées comme base dans différents domaines : la communication, l'interopérabilité et l'ingénierie des systèmes. À cela, il faut ajouter leur utilisation pour l'extraction et la manipulation de la connaissance ainsi que la multi-représentation des concepts.

2.3.1 Communication entre acteurs

Les ontologies permettent de spécifier le vocabulaire lié aux entrées-sorties d'acteurs [35]. Le vocabulaire utilisé est ainsi standardisé et classifié, ce qui permet d'éviter les ambiguïtés sémantiques lors de la communication. À l'instar de l'interface dans le paradigme objet, les ontologies décrivent un moyen de communication entre acteurs. Les ontologies se présentent comme le support d'un format d'échange et permettent l'interopérabilité des systèmes. Les acteurs peuvent alors coopérer en partageant les mêmes concepts, objets, méthodes, etc. Les auteurs de l'article [82] expliquent que l'utilisation d'un intermédiaire de traduction de type ontologie réduit le nombre de traducteurs nécessaires de $O(n^2)$ à $O(n)$.

Lorsque l'on parle d'interopérabilité entre les systèmes, il s'agit de communications et d'échanges d'informations afin d'atteindre un objectif donné. L'hétérogénéité des systèmes crée des différences de modélisation d'une information et rend la compréhension de cette information difficile ou ambiguë. L'interopérabilité entre les systèmes a comme objectif de traiter de manière cohérente les informations échangées. Pour réaliser l'interopérabilité entre des systèmes utilisant des ontologies différentes, il faut créer des liens entre les concepts des ontologies de chacun de ces systèmes. Cette recherche de lien s'appelle, dans la littérature, l'alignement des ontologies.

2.3.2 Multi-représentation

Les ontologies permettent de prendre en compte différents points de vue, autant lors de sa création que lors de son utilisation. Pour un acteur, le point de vue est la partie pertinente du domaine de l'ontologie qu'il utilise, c'est un sous-domaine. Un concept similaire existe dans le domaine des bases de données, les vues. Elles permettent de présenter une table virtuelle qui correspond à une extraction des données d'autres tables.

La création d'une ontologie d'un système complexe peut être réalisée en divisant le domaine d'un système en plusieurs sous-domaines. Ces sous-domaines contiennent une partie de la connaissance et forment une hiérarchie de dépendance.

2.3.3 Extraction et manipulation de connaissances

Les ontologies formalisent les connaissances d'un domaine en créant une hiérarchie entre les concepts du domaine et en contraignant les relations entre ces concepts. La formalisation de cette connaissance permet à des entités algorithmiques de manipuler les concepts du domaine et de se rapprocher de la compréhension de ces concepts telle que le fait un humain. En pratique, les ontologies formelles sont définies au moyen de langages proches de la logique du premier ordre.

D'une certaine manière, les modèles conceptuels sont différents des ontologies dans le sens où il n'est pas possible de vérifier la consistance du modèle et qu'il n'y a pas de possibilité d'inférer de nouvelles connaissances. De plus, les modèles conceptuels sont figés et difficilement accessibles lors de l'exécution du système. Enfin, les ontologies peuvent être vues comme une généralisation des modèles relationnels [35].

2.3.4 Ingénierie de systèmes

Les ontologies sont aussi utilisées dans l'ingénierie des systèmes d'informations d'entreprises [36]. Dans le cas où une ontologie du domaine d'une entreprise est disponible, celle-ci permet de réduire les couts d'analyse d'une application et permet de rendre cette application compatible avec l'ontologie. L'application développée bénéficiera ainsi des avantages des ontologies. Si l'ontologie du domaine n'existe pas, des ontologies de haut niveau (top-level ontology) sont disponibles et permettent d'augmenter la qualité du processus d'analyse. L'ingénierie des systèmes basés sur des ontologies est présentée de manière plus approfondie dans la section suivante.

2.4 Utilisation des ontologies en entreprise

Les méthodes et les infrastructures logicielles peuvent être étendues au moyen des ontologies. Elles bénéficient alors des qualités intrinsèques aux ontologies présentées dans la section précédente. L'aspect communication entre acteurs, l'extraction de la connaissance et la multi-représentation sont des propriétés que l'on retrouve souvent dans les logiciels où le domaine sous-jacent est complexe. De plus, un logiciel doit souvent pouvoir être extensible et permettre différentes formes d'interaction. Avant de présenter les méthodes d'exploitation, cette section présente les techniques de création des ontologies.

2.4.1 Création des ontologies

La création d'ontologies est un processus impliquant de nombreuses personnes de différents domaines, ce processus est donc très cher. Elle peut être appuyée par des techniques d'analyse de texte ou de données semi-formalisées pour créer une première ontologie, celle-ci devant être raffinée par la suite. En effet, les techniques d'extraction de connaissance automatique ne sont pas, pour le moment, un processus fiable et nécessitent une intervention humaine pour être valables. Une première technique est d'extraire un maximum d'informations d'un ensemble de données brutes (texte, HTML, XML, ...) ; c'est la technique qu'utilise l'outil Text2Onto

[13]. Une deuxième technique est de créer l'ontologie à partir d'un ensemble de documents et d'aider l'utilisateur à raffiner l'ontologie par itérations successives. Par exemple, les auteurs de [24] présentent l'outil OntoGen qui permet de construire une ontologie semi automatiquement au moyen d'une analyse des textes et d'interventions utilisateur. Ces techniques semi-automatiques ont pour ambition de réduire le coût de la création d'une ontologie, mais aussi de diminuer la charge de travail lors de la maintenance des ontologies. Enfin, les ontologies peuvent être créées à partir de zéro, à la main, en utilisant des outils d'éditions d'ontologies comme Protégé [3].

De nombreuses ontologies ont été créées pour des besoins industriels, et des méthodologies de création commencent à être de plus en plus précises. La méthode KASO [84] qui utilise la sortie de l'outil Text2Onto (outils d'extraction automatique de connaissances) permet d'organiser les concepts et de créer une ontologie par itérations successives. L'article [79] propose quant à lui une méthodologie de création d'ontologies intervenant à partir du début du processus de réalisation de l'ontologie. De plus, les auteurs expliquent la méthode de création d'une l'ontologie en faisant un parallèle avec le développement d'un logiciel.

2.4.2 Génie logiciel basé sur des ontologies

Les ontologies et les modèles conceptuels du génie logiciel partagent de nombreuses caractéristiques. Les deux représentent une abstraction d'un domaine et nécessitent une phase de conceptualisation qui consiste à retenir les éléments pertinents et exclure les éléments les moins importants d'un domaine. En revanche, il existe quelques différences présentées dans l'article [4] dont voici les plus importantes de notre point de vue :

- La création d'une ontologie est axée sur la prise en compte des entités (classes des concepts) structurant un domaine alors que la création d'un modèle logiciel est focalisée sur les opérations liées à des classes ;
- Les systèmes basés sur une ontologie peuvent interroger les données du modèle ainsi que le modèle. À l'inverse, la construction d'un logiciel autour d'un modèle ne permet pas d'interroger ce modèle, mais seulement les données. De plus, l'usage des ontologies permet d'étendre les différents modèles créés par la méthode d'ingénierie dirigée par les modèles ;
- Enfin, par leur utilisation, les modèles logiciels font la supposition que le domaine représenté est fermé, c'est-à-dire que tous les éléments nécessaires à l'exécution d'un programme sont modélisés. À l'inverse, les ontologies ne font pas cette supposition et permettent l'évolution du modèle.

3. http://protege.stanford.edu/

Les ontologies peuvent jouer un rôle important dans le génie logiciel. En 2006 les auteurs de l'article [43] identifient deux propriétés d'applications utilisant les ontologies. La première propriété permet d'identifier si l'ontologie utilisée à un rôle lors la phase de développement ou lors de l'exécution de l'application. La seconde permet de distinguer quel type d'apport l'utilisation des ontologies donne à un logiciel. L'apport pouvant être à un niveau technique ou à un niveau cœur de métier. Le croisement de ces deux propriétés donne quatre catégories d'utilisation :

- Le développement dirigé par des ontologies où l'ontologie est utilisée lors de la phase de développement en modélisant le cœur de métier. Dans cette optique, l'organisme W3C essaie de spécifier ce type d'utilisation dans le document [81] ;
- Le développement aidé par les ontologies, où les ontologies sont utilisées pour aider les développeurs dans leurs tâches. Par exemple la gestion de la phase de maintenance présentée dans [56] ;
- Les applications basées sur une ontologie : celles-ci utilisent une ontologie pour modéliser le cœur du domaine et pour accéder au modèle lors de l'exécution :
- Les applications aidées d'ontologies : celles-ci utilisent une ontologie comme un composant qui permet de résoudre un problème technique. Par exemple en utilisant l'aspect de communication entre acteurs présentée dans la section 2.3.1.

De ce constat, de nombreux auteurs ont proposé d'inclure des ontologies dans le processus de développement d'un logiciel.

Par exemple :

- Lee et Gandhi expliquent dans [60] une méthodologie qui facilite la découverte des exigences et l'écriture des spécifications en utilisant les ontologies ;
- dans [56] les auteurs présentent une ontologie qui permet d'aider la phase de maintenance d'un logiciel ;
- ou encore, l'auteur de l'article [58] présente une application orientée WEB exploitant une ontologie.

Ce dernier point, qui consiste à utiliser les ontologies dans un composant d'une application, est tellement important que l'organisme « Object Management Group » a créé une spécification sur la mise au point d'ontologies dans un système [32].

2.5 Outils pour les ontologies

La problématique de la création ou de l'exploitation des ontologies par les logiciels ne se limite pas à la méthodologie, il faut aussi choisir des outils et des techniques adaptés. Dans un premier temps, il faut choisir entre les différentes techniques de représentation des ontologies. Une fois le choix de la représentation effectué, il faut se poser la question du langage à utiliser pour stocker l'ontologie.

Enfin, l'ingénierie des ontologies nécessite des outils afin de créer, maintenir, stocker et utiliser les ontologies. Ces derniers peuvent être sous la forme de logiciel autonome, de composant logiciel, etc. Les éditeurs d'ontologies permettent de s'occuper des aspects création et maintenance alors que l'aspect exploitation est réalisé par des frameworks ou composants logiciels.

2.5.1 Techniques de représentation

Les techniques de représentation des ontologies permettent de capturer l'organisation et la structure de la connaissance humaine. Ces dernières utilisent le plus fréquemment des représentations à base de logiques ou de graphes.

La logique

La connaissance peut se représenter au moyen d'un langage logique. Le philosophe Aristote l'avait compris et inclus dans sont travail de réflexion dès les années 350 avant Jésus-Christ. C'est le domaine de l'intelligence artificielle qui utilisa le premier les langages logiques pour décrire des connaissances et les exploiter par des éléments calculatoires. De plus, les langages logiques possèdent une syntaxe qui permet d'écrire des « assertions » représentant la connaissance, ainsi qu'une possibilité d'inférence de connaissances. Une assertion désigne une phrase considérée comme vraie pour un domaine.

Les langages logiques donnent les moyens de décrire précisément une connaissance et éliminer les expressions incomplètes et ambiguës du langage naturel. Le corolaire de cette précision est qu'un raisonnement se basant sur un langage logique se limite à dériver les assertions modélisant un domaine. Il est difficile de modéliser le raisonnement humain qui fait intervenir, par exemple, les suppositions, les doutes ou les désirs.

Logique de proposition
Une description d'un domaine est composée de plusieurs propositions où chaque proposition est considérée comme une unité élémentaire de sens. Une proposition est un énoncé déclaratif (la fenêtre est ouverte), elle peut être soit vraie soit fausse. L'énoncé déclaratif est à opposer aux énoncés prescriptifs (un ordre : ferme la

fenêtre) ou aux énoncés intensionels (je pense que, je crois que...). Dans une description, les propositions peuvent être liées au moyen d'opérateurs logiques et former ainsi des propositions complexes ou des règles.

La logique des propositions utilise les propositions en tant que variables symboliques booléennes. La valeur logique de ces variables correspond à la vérité de la proposition sur le monde. Par exemple, si la variable p correspond à la proposition « mes fenêtres sont ouvertes », p est vrai si toutes mes fenêtres sont ouvertes dans le monde décrit. La description d'un monde ne se limite pas aux propositions, mais fait intervenir des règles. Les règles peuvent être vus comme la description d'une conséquence d'une proposition : si p alors e. Dans la logique des propositions les règles sont décrites par l'implication $p \implies e$. Ce qui correspond à dire que e est obligatoirement vrai lorsque p est vrai, par contre e peut être vrai si p est faux. Une proposition liée à l'implication suit la règle booléenne suivante : $\neg p \vee e$.

Cette logique permet de raisonner sur des symboles en combinant les valeurs des variables symboliques avec les règles. Cela signifie que l'on utilise la syntaxe de la logique booléenne et non la signification des propositions pour faire les calculs de raisonnement.

La puissance de description de la logique des propositions est étendue au moyen de la logique du premier ordre.

Logique du premier ordre

La logique du premier ordre étend la puissance de description de la logique des propositions. Elle introduit aussi le quantificateur d'existence $\exists$, le quantificateur universel $\forall$ ainsi que l'utilisation de variables et de prédicats. Un prédicat représente une propriété d'un objet ou une relation entre plusieurs objets. Kurt Gödel a prouvé dans sa thèse en 1929 que la logique du premier ordre est complète. Cela signifie que cette logique permet de démontrer qu'une proposition est vraie ou fausse dans un système composé d'axiomes et de règles.

Cette logique forme une base solide pour représenter la connaissance sur laquelle le domaine de l'intelligence artificielle a créé de nombreux langages, comme Prolog. La connaissance est représentée, dans cette logique, sous forme d'axiomes et de règles. Les axiomes sont décrits par un ensemble de constantes et de prédicats.

La logique du premier ordre peut-être utilisée pour représenter les ontologies, car elle décrit les liens logiques entre les propositions et permet l'inférence des connaissances automatiquement, notamment au moyen du *modus ponens*. Mais elle reste un langage de représentation de la connaissance bas niveau dans le même sens que l'assembleur est un langage de programmation bas-niveau. Le langage Knowledge Interchange Format (KIF) [26] est un bon exemple. Ce dernier est une extension directe de la logique du premier ordre, son but est d'être un moyen

d'échange de connaissances entre différents systèmes et n'est pas destiné à être utilisé pour raisonner.

Si la logique du premier ordre est complète, elle est indécidable du point de vue algorithmique. En effet, le nombre d'étapes pour démontrer une proposition n'est pas borné dans la logique du premier ordre. Dans ces circonstances, une nouvelle forme de représentation de la connaissance a été mise au point : la logique de description qui constitue un sous-ensemble de la logique du premier ordre.

Logique de description

La logique de description (DL) se place entre la logique des propositions, car elle est plus expressive, et la logique du premier ordre, car elle est plus efficace dans la démonstration des propositions. DL convient parfaitement à la description d'une ontologie, car elle permet la description formelle de la connaissance et peut être outillée afin de raisonner [5].

DL introduit deux composants appelés Terminological Box (TBox) et Assertion Box (ABox). Ces composants peuvent être vus comme des conteneurs permettant de stocker des éléments de connaissance. Les concepts, leurs propriétés et leurs relations sont stockés dans les TBox, quant aux entités et réalisations, elles sont stockées dans les ABox. En résumé, TBox définit les concepts en intension alors que ABox définit les concepts en extension.

DL décrit les concepts par unité de connaissance et par composition de ces dernières. Une unité de connaissance correspond à un élément de connaissance indivisible ou encore atomique. Ainsi, la TBox stocke des concepts et des relations atomiques ainsi que des concepts et relations complexes. Reprenons l'exemple de l'article [5] ; *Femelle* et *Humain* sont deux concepts atomiques et *aEnfant* est une relation atomique.

À partir de ces deux concepts, nous pouvons indiqué de manière informelle une règle pour composer des concepts complexes comme *Femme*, *Homme* ou *Père* :

- Femme $\equiv$ Humain $\cap$ Femmelle : une femme est une humaine et une femelle, c'est-à-dire une humaine du sexe féminin ;
- Homme $\equiv$ Humain $\cap$ $\neg$Femme : un homme est un humain qui n'est pas une femme ;
- Pere $\equiv$ Homme $\cap$ $\exists$ aEnfant : un Père est un homme ayant un enfant.

Ainsi, la définition des entités dans la ABox pourrait être :

$$\text{Femme}(\text{AURORE})$$
$$\text{aEnfant}(\text{AURORE}, \text{BOB})$$

Il existe plusieurs formes de DL, leur but étant de permettre le raisonnement et de décider si un prédicat donné est vrai ou faux. La décidabilité et la complexité du calcul d'inférence dépendent de la puissance d'expression de la DL. L'inférence d'une DL avec un fort pouvoir d'expression est très complexe et risque d'être indécidable. À l'inverse, une DL avec peu d'expressivité sera plus efficace pour l'inférence de connaissances, mais celle-ci risque de ne pas pouvoir décrire entièrement le domaine d'une application. Il faut donc trouver un compromis entre la puissance de description et la décidabilité, et chercher à améliorer ces deux éléments.

DL est probablement la méthode de description la plus utilisée pour modéliser les ontologies, notamment au moyen du langage OWL décrit dans la section 2.5.2.

Les graphes

La représentation de la connaissance sous forme de graphe est intuitive et il en existe de nombreuses formes. Les graphes permettent de représenter les connaissances sous une forme visuelle très simplement. Les nœuds représentent les objets du monde (concept, relation, entité, ...) et les arcs des liens entre les objets. De plus, certaines représentations sous forme de graphe représentent la logique du premier ordre ou un sous-ensemble formel de cette logique, ce qui permet d'exploiter les graphes au moyen d'un élément algorithmique de raisonnement et d'inférence.

Les graphes conceptuels sont l'une de ces représentations ; ils permettent de décrire les concepts et leurs relations. Ils ont été introduits par Sowa en 1976 dans un article les présentant comme une interface entre le langage naturel et une base de données [75]. Ces graphes sont bipartis et contiennent deux types de nœuds : les concepts et les relations. Les arcs entre les nœuds sont dirigés et se font toujours entre deux types de nœuds différents, c'est à dire qu'un concept ne peut pas être lié à un concept, de même qu'une relation ne peut pas être liée à une relation. Les graphes conceptuels ont la particularité de représenter les propositions de manière graphique ainsi que d'être construits sur une logique du premier ordre.

Les graphes conceptuels sont un domaine de recherche en activité et leur formalisation est étendue et évolue continuellement. De ce fait, deux standards sont apparus afin d'apporter la stabilité d'une norme et promouvoir les graphes conceptuels. Ces deux normes sont la norme « ISO standard for Common Logic » [48] ainsi que la norme « IKL Conceptual Graphs » [44] qui étend la première. Ces documents donnent des règles qui permettent de raisonner sur les graphes conceptuels.

2.5.2 Langages

La section précédente a présenté deux bases (logique ou graphe) permettant de formaliser la représentation de la connaissance. Cette section indique quels sont les langages permettant de stocker et manipuler les ontologies.

Langages des ontologies

Il existe de nombreux langages permettant de représenter une ontologie. Nous pouvons les classer en deux catégories, ceux qui sont nés avant l'apparition du XML et les autres. Les premiers sont apparus dans les années 1990 pour résoudre des problèmes de représentation de la connaissance dans le domaine de l'intelligence artificielle. Voici deux exemples des langages les plus connus de la première catégorie :

- KIF [26], basé sur la logique du premier ordre ;
- Ontolingua [33].

Les seconds sont apparus à partir de la fin des années 1990 et sont, pour la plus part, décrits avec le langage XML. Ces langages de nouvelles générations sont mis au point avec l'intention de créer un WEB où la connaissance diffusée sur ce média serait utilisable par les machines. Cette nouvelle ère se nomme le WEB sémantique [72].

Un langage de stockage peut représenter un domaine comme un monde ouvert ou comme un monde fermé. La représentation en monde ouvert signifie que de nouveaux éléments de connaissance peuvent être ajoutés au fur et à mesure, en d'autres termes que le domaine n'est pas entièrement caractérisé. À l'inverse, la représentation en monde fermé signifie que le périmètre de la connaissance est défini à priori et que tous les éléments utiles à la compréhension d'un domaine sont présents.

Quelques langages de représentation des ontologies ont été conçus pour Internet et spécialement pour le Web. Parmi eux, nous pouvons cité OIL et DAM+OIL qui sont tous les deux les prédécesseurs du langage OWL (Ontology Web Language). Notons que OIL et DAM+OIL sont deux langages qui correspondent à la représentation sous forme de logique de description. L'article [45] explique les différences entre les langages précurseurs de OWL et les différentes méthodes de représentation qui ont été utilisées pour créer le langage OWL.

OWL permet de structurer les ontologies avec un dialecte reposant sur le langage de balise XML. Il existe trois dérivés du langage OWL : Lite, DL et Full. OWL Lite et OWL DL reprennent les principes de la logique de description. Par contre, OWL Full ne reprend pas les principes de la logique de description [8]. En réalité, OWL Full n'impose pas autant de contraintes que OWL DL. Par exemple

il est possible d'utiliser les classes comme des instances d'autres classes, ce qui empêche la séparation des éléments de TBox (les classes) et les éléments de ABox (les instances). Cette séparation est nécessaire pour utiliser la représentation de la connaissance au moyen de la logique de description décrite dans la section 2.5.1. Dans la suite de ce manuscrit, nous ne parlerons plus de OWL Full.

La principale différence entre OWL Lite et DL est le choix du compromis entre la décidabilité et l'expressivité du langage. OWL DL reprend les grandes lignes de la logique de description. OWL Lite est un sous-ensemble de OWL DL. Il est utilisé lorsque la rapidité de décision prime sur l'expressivité du langage. Le fait que OWL corresponde à la logique de description (DL) lui permet de bénéficier des résultats des recherches sur l'inférence ainsi que des outils qui ont été développés pour raisonner. Par exemple, l'outil Pellet [73] peut être utilisé comme moteur d'inférence pour les connaissances exprimées en OWL.

OWL fait l'hypothèse d'un monde ouvert et permet de décomposer l'ontologie en plusieurs fichiers. De plus, chacun des fichiers contient les connaissances qui lui sont nécessaires pour être auto-suffisant et il représente une sous-partie de l'ontologie.

SPARQL

SPARQL est aux composants de stockages de connaissances ce que le langage SQL est aux bases de données. Ce langage commence à s'imposer comme le langage de requête des composants de stockage de connaissances. En effet, il est depuis 2008 une recommandation W3C [4]. Il permet d'extraire les informations correspondant à une requête exprimée sous forme de propositions et de contraintes. Par contre, la modification de la connaissance lors de l'exécution n'est pas aussi évidente que le changement dans une base de données. Une version 1.1 est en cours d'élaboration est permettra de modifier cette connaissance [5] au moyen de SPARQL.

2.5.3 Éditeurs d'ontologies

Les langages qui modélisent les ontologies utilisent, pour la plupart, une description en XML. Sa création et son édition peuvent être faites à la main avec un simple éditeur de texte ou un éditeur XML, en revanche cette opération est fastidieuse et sujette à créer des erreurs. Des outils plus élaborés existent et permettent de représenter une vue d'une ontologie sous forme d'arbres ou de graphes.

4. Recommandation W3C à l'adresse : http ://www.w3.org/TR/rdf-sparql-query/. Dernier accès novembre 2011.

5. http ://www.w3.org/TR/2011/WD-sparql11-query-20110512/ Dernier accès novembre 2011

L'auteur de l'article [12] présente le taux d'utilisation de treize éditeurs. D'après l'étude de cet article, l'outil le plus populaire est sans nul doute Protégé.

Protégé [27] est un éditeur d'ontologie possédant sa propre représentation interne de la connaissance. Il représente les concepts et les entités des ontologies en deux arbres (un arbre-concepts et un arbre-entités), mais cela reste une représentation. Comme l'ontologie est un graphe, Protégé permet à une sous-branche d'un arbre d'apparaître dans plusieurs nœuds. Protégé dispose d'un système de plug-ins qui permet d'étendre ses possibilités et ses fonctionnalités. Par exemple, il existe une extension (OntoViz) qui donne les moyens de visualiser graphiquement l'ontologie en cours d'édition. D'autres plug-ins permettent d'importer et exporter une ontologie en différents formats de fichiers, dont OWL. Il donne également la possibilité d'utiliser des raisonneurs pour vérifier qu'une ontologie est cohérente et que les contraintes syntaxiques sont respectées.

Pour plus d'informations sur les éditeurs d'ontologies, le lecteur peut se référer à l'article [63]. Ce dernier présente quatre autres éditeurs en plus de Protégé et compare leurs caractéristiques et fonctionnalités.

2.5.4 Composants d'exploitation

L'utilisation des ontologies n'est pas limitée au choix d'une représentation et d'un langage, il faut des outils d'exploitation. Il existe des bibliothèques de programmation (API) et des moteurs qui fournissent les moyens de charger, manipuler et interroger les ontologies. D'autres composants, les raisonneurs, permettent de vérifier les contraintes imposées par les propriétés des relations ainsi que d'inférer de nouvelles connaissances.

Dans les cas les plus simples, le stockage de l'ontologie se fait sous forme de fichiers. Mais lorsque l'application a des exigences de mises à l'échelle, de performance et ou autres, cette solution atteint ses limites. En effet, lire les fichiers d'ontologie est très consommateur en temps et charger en mémoire une ontologie n'est pas toujours possible. Dans les cas plus complexes, il existe des mécanismes qui permettent de stocker les ontologies sous forme de triplet [86]. Ces solutions donnent alors différents moyens d'accès à la base de connaissances : requêtes avec un langage spécialisé (SPARQL), SOAP, requête HTTP, etc.

L'article [86] présente le moteur d'ontologie Jena2. Ce dernier propose de stocker les ontologies sous forme de triplet dans une base de données et optimise l'accès aux éléments de connaissance. Il permet d'utiliser le langage SPARQL pour chercher des éléments de connaissance. L'application OWL2Java complète Jena. Celui-ci génère des classes correspondant au modèle défini dans une ontologie. Les classes correspondent aux concepts et les attributs des classes correspondent aux

propriétés des concepts. Cette technique permet d'incorporer le modèle de l'ontologie directement dans l'application. Jena dispose de quelques algorithmes en interne pour inférer sur les ontologies. Des raisonneurs externes, plus complets, peuvent être utilisés.

Une autre approche est d'utiliser les ontologies comme un composant de l'application et non comme le coeur de l'application [43]. OWL API est une bibliothèque de programmation (API) qui permet d'accéder à une ontologie stockée en format OWL. Ainsi, le modèle de l'ontologie est accessible de la même manière que les instances de ses concepts. De plus, OWL API permet d'interfacer des raisonneurs externes.

2.6 Synthèse

Ce chapitre a introduit les ontologies, le rôle qu'elles peuvent tenir au sein des applications logicielles et les outils permettant de les exploiter. L'utilisation des ontologies permet de définir un vocabulaire commun et améliore la communication entre acteurs. Elle permet aussi d'utiliser un concept sous différentes représentations. Les ontologies sont étudiées depuis les années 1990 et de nombreux outils sont disponibles pour les exploiter. Plusieurs définitions du terme sont disponibles dans la littérature, mais la suite de ce mémoire utilise le terme ontologie pour se référer à une base de connaissances formalisée utilisable par un composant logiciel. Dans ce chapitre, nous avons considéré le format de stockage de la connaissance OWL ainsi que quelques outils disponibles pour ce format.

L'étude du comportement de la maison nécessite de décrire l'évolution de systèmes. Pour cela, nous utilisons un autre outil permettant de manipuler le comportement des objets communicants. Cet outil se base sur la théorie des graphes, plus précisément les automates étendus. Ainsi nous dissocions les connaissances du modèle de la maison du comportement des dispositifs.

Sommaire

Chapitre 3

Le modèle de comportement

Les chapitres précédents ainsi que le chapitre 4 mettent en évidence le problème de commandabilité des dispositifs. À la réception de commandes, un dispositif réagit et produit un comportement. Les technologies présentées dans les chapitres précédents ne sont pas suffisantes pour modéliser ce type de comportement. En effet, ces technologies ne permettent pas à elles seules de trouver les séquences de commandes à envoyer à un dispositif afin de lui donner l'état désiré. C'est à l'aide d'un autre mécanisme que nous allons modéliser le comportement des dispositifs afin de prendre en compte les transitions (c'est-à-dire un ensemble ordonné de commandes) entre les états.

3.1 Choix du modèle de comportement

Un automate, ou système de transitions est un outil mathématique utilisé dans le domaine de l'informatique. Il permet de modéliser des systèmes ou des calculs afin de mettre en évidence certaines propriétés de ces éléments. La théorie des automates est utilisée :

- Dans le domaine de la technologie, en particulier dans l'automatisme et le contrôle de systèmes. L'étude des automates a permis de développer des méthodes d'analyse et de synthèse des modèles de systèmes. Ces méthodes permettent de vérifier si un modèle correspond à des propriétés de sécurités et si un système physique correspond à un modèle.
- Dans le domaine de l'informatique fondamentale, en particulier pour l'étude

des algorithmes et des langages. Elle permet de répondre aux questions sur le problème de décidabilité. C'est-à-dire qu'elle est la complexité d'un algorithme en terme d'espace et de temps.

Un automate est un graphe orienté ou les sommets sont les configurations d'un système et les arcs sont ses transitions. Un graphe est constitué d'un ensemble de sommets, d'un ensemble de liens et d'une fonction d'incidence qui associe chacun des liens à deux sommets. Un graphe est dit orienté lorsque les liens sont orientés, c'est-à-dire que l'association de deux sommets a et b ne se fait que pour la paire $\langle a, b \rangle$. À l'inverse, la fonction d'incidence d'un graphe non orienté fait l'association de deux sommets a et b pour l'ensemble $\{a, b\}$.

Par nature, les systèmes sont dynamiques, c'est-à-dire que le changement de configuration est réalisé d'une façon continue. Même une lampe qui possède deux états, allumé et éteint, doit subir un changement progressif de température du filament qui la compose. Les automates permettent de modéliser les caractéristiques utiles des systèmes et de supprimer les autres. Les automates sont de nature discrète ou hybride (c'est-à-dire à la fois discret et dynamique).

Un automate discret prend en compte les états d'un système de manière symbolique. Pour le cas de la lampe, les états symboliques sont « allumé » et « éteint ». Un automate hybride prend en compte l'évolution du système de chacun des états. L'évolution du système est caractérisée par une ou plusieurs équations mathématiques. C'est le cas de la luminosité créée par une lampe lors du passage de l'état froid à l'état chaud du filament.

Les dispositifs domotiques peuvent entraîner l'utilisation de nombreuses configurations, voire même d'une infinité. Déterminer le comportement d'un système infini par un automate dit infini est un problème complexe si l'on ne fait pas intervenir des simplifications. En l'occurrence, il faut calculer, à partir d'un ensemble de départ, l'ensemble des configurations accessibles en une étape, puis en un nombre d'étapes quelconque. La modélisation d'un système infini fait intervenir un mécanisme de regroupement d'états au sein des sommets. Informellement, cette opération consiste à replier un automate infini dans un automate fini comportant la même sémantique (c'est-à-dire le même nombre d'états possibles et les mêmes transitions entre les états). Les sommets du graphe de cet automate fini sont appelés localités.

Le besoin est de modéliser le comportement de dispositifs de la maison. Ce comportement à comme caractéristiques :

1. Pour un système extérieur observant, ces dispositifs sont de nature discrète, c'est-à-dire que l'évolution d'un état à un autre se réalise en un temps non mesurable. La seule information sur l'évolution des dispositifs est envoyée par des notifications.

2. De gérer les commandes utilisables par un dispositif et les notifications que produit ce dernier.
3. De disposer d'un nombre infini d'états, par exemple une lampe ajustable possède d'une gamme de luminosité entre 0% et 100%.
4. De gérer les paramètres des commandes et des notifications dont nous parlons dans le chapitre suivant.
5. De pouvoir évoluer de manière autonome, une des particularités des environnements domotiques est le nombre de sources de contrôle. En effet, chaque acteurs d'un système domotique à un accès direct aux commandes des objets communicants disponibles dans l'environnement. C'est pourquoi une réponse qu'un dispositif envoie n'est pas toujours attendue ou n'est pas toujours celle qui est attendue.

Ces caractéristiques nous dictent le choix à effectuer afin de modéliser les systèmes domotiques. La première propriété nous propose l'utilisation d'automates discrets. Les automates hybrides sont plus complets, mais la dynamique du système n'apporte pas d'informations utiles au besoin de modéliser la commandabilité des dispositifs. La deuxième caractéristique nous indique de prendre un modèle d'automates où les transitions portent un sens. Il faut donc utiliser des automates de la famille des systèmes de transitions étiquetés. La troisième propriété implique l'utilisation d'automate où les sommets ne représentent plus un unique état, mais un ensemble d'états. Ces sommets sont appelés des localités. Les automates qui en découlent sont nommés systèmes de transitions symboliques. De plus, il faut un outil mathématique permettant d'explorer les configurations d'un système infini. Ce type d'automate permet de prendre en compte la quatrième caractéristique puisque les transitions ont elles aussi un sens et sont représentées par des symboles. Et enfin, la cinquième propriété impose de trouver un mécanisme qui prend en compte la possibilité d'évolution du dispositif.

Les automates IOSTS (*Input/Output Symbolic Transitions System*) prennent en compte ces différentes caractéristiques.

Il est bon de noter qu'il existe un langage formel qui permet de décrire un type de diagramme d'états, le langage SCXML [83]. Ces diagrammes d'états ressemblent aux diagrammes d'états de la norme UML. Ils permettent de modéliser une machine à états complexe. En revanche, ce modèle manque de formalisation mathématique.

L'utilisation d'un modèle basé sur les systèmes de transitions nous permet de bénéficier de la littérature dans le domaine des automates. Notamment des outils pour vérifier et tester les systèmes.

3.2 Systèmes de transition

La théorie des systèmes de transitions à entrées et sorties symboliques (IOSTS pour *Input/Output Symbolic Transitions System*) est utilisée dans la littérature à la fois pour vérifier, tester et contrôler des systèmes. La vérification et le test sont des techniques formelles qui permettent de valider un dispositif. Ces deux techniques consistent à comparer deux vues d'un même système [16]. Le contrôle permet de contraindre le comportement de systèmes.

La vérification formelle est un mécanisme qui compare un modèle formel du comportement d'un système avec des propriétés que ce système doit satisfaire. Le test fonctionnel quant à lui, s'effectue lors de l'exécution du système. Celui-ci compare des cas de test issus d'un modèle formel avec le comportement d'un système réel lors de l'exécution de ces cas de test. Ce qui signifie qu'un test peut détecter seulement une différence entre le cas de test et le système, il ne peut pas prouver qu'un système est conforme au modèle.

Ces deux techniques se basent sur un modèle théorique du système, mais le modèle utilisé pour le test n'est pas toujours compatible avec le modèle pour vérifier. Or, ces techniques sont complémentaires. La plus-part du temps, le système réel est créé à partir d'un modèle qui a été vérifié. Le test intervient une fois que le système est créé, il permet de vérifier que le système réel est conforme au modèle. L'utilisation du test fait l'hypothèse que la spécification est conforme au comportement attendu. Or cette hypothèse est fausse dans le cas où les cas de tests n'ont pas de lien formel avec les propriétés vérifiées d'un modèle. La thèse de C. Constant [17] prend en compte ces problèmes et fournit des outils de génération de tests en fonction de propriétés que l'on souhaite vérifier.

3.2.1 Définition formelle

Contrairement à la définition du modèle IOSTS de [17], nous utilisons ici la notion explicite des localités comme c'est le cas dans les articles [15, 16]. Par la suite, nous donnerons différentes propriétés aux localités, ces propriétés sont liées au comportement des objets communicants. L'idée proposée par C. Constant [17] est d'énumérer les localités par le domaine d'une variable. Les structures de contrôle se réfèrent alors à la valeur d'une variable particulière au lieu de se référer à une localité.

Definition 1 *Un IOSTS est un tuple* $\langle D, \Theta, L, l_0, \Sigma, T \rangle$ *tel que :*

- $D = V \cup P$ *est un ensemble fini de données, partitionnée d'un ensemble fini* V *de variables et d'un ensemble fini* P *de paramètres. Soit un élément* $d \in D$*, Dom(d) détermine le domaine de valeur de* d*. Si* $e \in \mathrm{Dom}(d)$*, alors* e *est une valeur compatible avec* d*.*

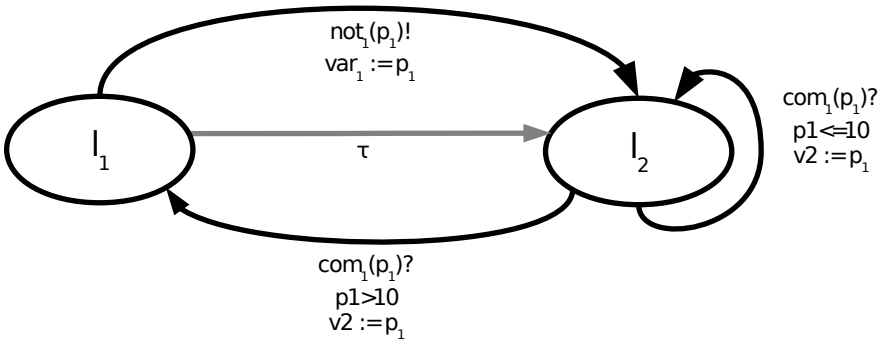


FIGURE 3.1 – Représentation graphique d'un IOSTS

- Θ *est l'ensemble des conditions initiales, c'est un prédicat sur* V*.*
- L *est un ensemble fini non vide de localités et* $l_0 \in L$ *est la localité initiale. Une localité est un ensemble d'états* $l \subseteq Dom(V)$*.*
- Σ *est l'alphabet, un ensemble fini et non vide d'actions. Il est composé de l'union disjointe de l'ensemble* $\Sigma^?$ *d'actions d'entreé, de l'ensemble* $\Sigma^!$ *d'actions de sortie et de l'ensemble* Σ^τ *d'actions internes. Pour chaque action* $a \in \Sigma$*, sa signature* $sig(a) = \langle p1, ..., pk \rangle \in P^k (k \in \mathbb{N})$ *est un tuple de paramètres. La signature des actions internes est toujours un tuple vide.*
- T *est un ensemble de transitions fini, tel que chaque transition est un tuple* $t = \langle l^o, a, G, A, l^d \rangle$ *défini par :*
  - *une localité* $l^o \in L$*, appelée l'origine de la transition,*
  - *une action* $a \in \Sigma$*, appelée l'action de la transition,*
  - *un prédicat* G *sur* $V \cup sig(a)$*, appelé la garde,*
  - *une fonction d'affectation* $A : Dom(V) \mapsto Dom(V)$*, qui est un ensemble d'expression de la forme* $(x := A^x)_{x \in V}$ *tel que, pour tout* $x \in V$*,* $A^x \in V \cup sig(a)$*,*
  - *une localité* $l^d \in L$*, appelée la destination de la transition.*

La figure 3.1 illustre un automate IOSTS d'un système fictif ayant comme caractéristiques d'être à la fois simple et de comporter tous les éléments possibles. Cet automate est composé de deux localités $L = \{l_1, l_2\}$, trois actions, quatre transitions et deux variables. La définition formelle est :

- Les variables sont $V = \{var_1, var_2\}$ de type $\text{type}(var_{\{1,2\}}) = \mathbb{R}$.
- Les actions de l'alphabet de cet automate sont $\Sigma^? = \{com_1\}$, $\Sigma^! = \{not_1\}$ et $\Sigma^\tau = \{\tau\}$.

- Les signatures des actions est $\text{sig}(com_1) = \{p_1\}$, $\text{sig}(com_2) = \{p_2\}$ et $\text{sig}(\tau) = \emptyset$.
- Les transitions sont :
  - $t_1 = \langle l_1, not_1, true, var_1 := p1, l_2 \rangle$,
  - $t_2 = \langle l_2, com_1, p1 > 10, var_2 := p1, l_1 \rangle$,
  - $t_3 = \langle l_2, com_1, p1 <= 10, var_2 := p1, l_2 \rangle$,
  - $t_4 = \langle l_1, \tau, true, \emptyset, l_2 \rangle$,

L'automate de la figure 3.1 modélise un système infini. En effet, il existe une infinité d'états représentés par les variables var_1 et var_2 puisque leur domaine est l'ensemble des réels, $\text{type}(var_{\{1;2\}}) = \mathbb{R}$.

Ce qui est important dans la signature d'une action, c'est le nombre et l'ordre des paramètres et non l'étiquette qui est donné aux paramètres. Les paramètres des actions sont nommés localement à une action. Par exemple, $\text{sig}(com_1) = \{p_1\}$ indique qu'il y a un et un seul paramètre. Ce paramètre n'a rien a voir avec le paramètre p_1 de l'action not_1 donc, $sig(com_1) \neq sig(not_1)$.

Les gardes permettent de savoir si une transition peut être réalisée lorsqu'une action a lieu. Dans cet exemple, les transitions t_2 et t_3 disposent de la même action com_1 et ont la même origine l_2. Cette configuration induit un phénomène d'indécidabilité dans d'autres théories relatives aux automates. Dans notre cas, les gardes permettent d'indiquer quel est la transition à choisir lors de la réalisation de l'action com_1.

Les actions d'entrée ($\Sigma^?$) d'un système représentent les commandes émises par un point de commande. Ce sont les actions que ce système reçoit. Les actions de sortie ($\Sigma^!$) représentent les notifications que le système envoie. Du point de vu du point de commande, ce sont les actions reçues.

Les actions internes quant à elles représentent des actions dont il est impossible de savoir si elles ont été réalisées. L'évolution d'un dispositif est liée aux notifications qu'il émet et aux commandes qu'il reçoit, mais la prise en compte du média de communication implique une évolution silencieuse du système. Pour un point de commande, cela engendre une désynchronisation entre l'état supposé d'un système et l'état réel de ce système. Un point de commande ne peut pas connaître l'évolution d'un système lorsqu'une action interne se produit. Les actions internes sont intéressantes pour modéliser le problème d'évolution silencieuse, mais amènent un problème d'indéterminisation lors du test ou de la validation des systèmes modélisés. Pour prendre en compte ces actions internes, les modèles IOSTS doivent être déterminisés [51].

Soit $\mathcal{S} = \langle D_\mathcal{S}, \Theta_\mathcal{S}, L_\mathcal{S}, l^0_\mathcal{S}, \Sigma_\mathcal{S}, T_\mathcal{S} \rangle$ l'automate IOSTS d'un dispositif. Dans le cas du test et de la vérification, le fait qu'une transition n'apparaisse pas, $t \notin T_\mathcal{S}$,

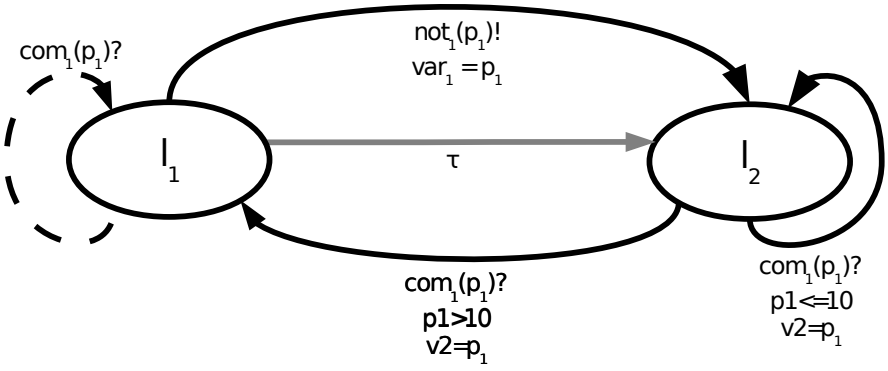


FIGURE 3.2 – Transition sous-entendue d'un objet fictif IOSTS

indique que l'action $a \in t$ ne peut pas être déclenchée et qu'il est mécaniquement impossible de tirer cette action. Dans le cas de la modélisation d'objets communicants, le fait que t n'apparaisse pas revient à une simplification d'écriture afin de ne pas faire apparaître des actions sans utilité, c'est-à-dire sans modification d'état du système. En effet, chaque objet communicant peut recevoir une commande, même si celle-ci n'est pas prise en compte dans l'état actuel du dispositif, le dispositif n'a d'autre choix que d'ignorer l'ordre.

Definition 2 *Formellement, les transitions de sortie des dispositifs jouissent de cette règle :*

- *Soit $T^a_{l_1 \to l\_}$ l'ensemble des transitions ayant pour source la localité l_1, comme destination une autre localité et comme action a.*
- *Soit $G^a_{l_1}$ l'ensemble des gardes des transitions en provenance de l_1 à destination d'une autre localité l_2 ayant pour action $a \in \Sigma$.*

$$\forall t \in T^a_{l_1 \to l\_} \subseteq T \mid l_1 \neq l_2 \wedge a \in \Sigma \wedge t =< l_1, a, g, , l_2 >, g \in G^a_{l_1}$$

- *Il existe une transition d'entrée d'action t^a en provenance de la localité l_1 à destination de elle même qui porte sur le complément des gardes des transition de $T^a_{l_1 \to l\_}$ et dont la fonction d'affectation est vide $b = \text{nop}$ (nop signifie* no opération*). Et cela, pour toutes les localités.*

$$\forall l_1 \in L, \forall a \in \Sigma^?, \exists t^a \in T \mid \{\bar{g}\} = \bar{G^a_{l_1}} \wedge b = \text{nop} \wedge t^a =< l_1, a, \bar{g}, b, l_1 >$$

Cette propriété nous permet de sous-entendre une transition d'entrée lorsqu'elle boucle sur elle-même et qu'elle n'apporte pas d'affectation b. Il est donc possible de ne pas faire apparaître cette transition puisqu'elle découle de cette propriété. Par exemple, si la figure 3.1 correspond à un dispositif domotique représenté en IOSTS,

la transition $t = \langle l_1, com_1, true, \text{nop}, l_1 \rangle$ est sous-entendue. La figure 3.2 illustre la transition sous-entendue du dispositif fictif si celui-ci est un objet communicant. La transition sous-entendue est représentée au moyen du trait discontinue sur la gauche.

De même, nous ne faisons pas apparaitre les gardes lorsque celles-ci sont toujours vraies $G = true$, ainsi que les affectations qui ne modifient pas la valeur des variables.

L'évolution de l'état d'un système est formulée sous forme d'exécutions [16]. Une exécution correspond à une séquence de transitions tirables. Une transition tirable est un tuple de trois éléments :

- état source de la transition,
- transition
- valeurs des paramètres des actions satisfaisant la garde de la transition.

Definition 3 *Un fragment d'exécution est une séquence alternant états et actions tirables de la forme* $s_1\lambda_1 s_2\lambda_2 \ldots \lambda_n s_n \in S \cdot (\Lambda \cdot S)^*$. *Une exécution est un fragment d'exécution démarrant depuis un état initial.*

On note Exec($\mathcal{S}$) l'ensemble des fragments d'exécution de l'automate IOSTS $\mathcal{S}$. Un état s est atteignable si cet état appartient à une exécution.

L'exécution est le comportement interne d'un système $\mathcal{S}$, il englobe tous les types d'actions et notamment les actions internes Λ^τ. Lorsqu'un système extérieur analyse le comportement de $\mathcal{S}$, celui-ci ne peut pas observer les actions internes. Il s'intéresse donc au comportement visible de $\mathcal{S}$, c'est-à-dire aux actions d'entreé et de sortie. On note ce comportement visible des traces [16]. Les traces peuvent être définies par rapport à une exécution.

Definition 4 *La trace Trace(ρ) d'une exécution ρ est la projection de ρ sur l'ensemble des actions observables* $\Lambda^? \cup \Lambda^!$. *L'ensemble des traces Traces($\mathcal{S}$) d'un système $\mathcal{S}$ est l'ensemble des traces des exécutions de $\mathcal{S}$*

3.2.2 Sémantique des automates

Un automate IOSTS est une syntaxe qui permet de décrire de manière finie des systèmes de transitions infinis. La sémantique d'un IOSTS, c'est-à-dire sa signification, peut être décrite au moyen d'un système de transitions à entrées et sorties étiquetées (IOLTS pour Input/Output Labelled Transitions Systemes) [16]. Informellement, la sémantique d'un IOSTS sous forme d'un IOLTS énumère tous les états possibles et toutes les transitions possibles.

Definition 5 *Un IOLTS est un tuple* $\langle S, S^0, \Lambda, \rightarrow \rangle$ *tel que :*

- S *est un ensemble non-vide d'états.*
- $S^0 \subseteq S$ *est l'ensemble non vide des états initiaux.*
- $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ *est l'ensemble non-vide des actions. Il est composé de l'ensemble des actions d'entrée* $\Lambda^?$*, de sortie* $\Lambda^!$ *et d'actions internes* Λ^τ*. Dans le contexte des automates IOLTS, les actions ne sont pas paramétrées.*
- $\rightarrow \subseteq S \times \Lambda \times S$ *est l'ensemble non-vide des transitions d'états.*

Afin d'obtenir la sémantique $\mathcal{H} = \text{IOLTS}[\mathcal{S}]$ d'un automate IOSTS $\mathcal{S}$, il faut explorer tous les états produits par le déclenchement de toutes les actions possibles à partir des localités initiales. Si $D' = \langle d_1, d_2, \ldots, d_n \rangle$ est un tuple de données de $d_{1,2,\ldots,n} \in D$, on note $\text{Dom}(D')$ le tuple du domaine de chacun des éléments de D'. C'est-à-dire, $\text{Dom}(D') = \langle \text{Dom}(d_1), \text{Dom}(d_2), \ldots, \text{Dom}(d_n) \rangle$. Autrement dit, $\text{Dom}(D')$ est l'ensemble des tuples de valeurs possibles des données D de l'IOSTS $\mathcal{S}$. L'élément $d \in \text{Dom}(D')$ est un tuple composé de valeurs affectées aux variables de D', il s'agit de l'affectation de D'.

Definition 6 *La sémantique d'un IOSTS [16]* $\mathcal{S} = \langle P \cup V, \Theta, L, l_0, \Sigma, T \rangle$ *est un IOLTS* $\mathcal{H} = \langle S, S^0, \Lambda, \rightarrow \rangle$ *défini par :*

- *l'ensemble des états* $S = L \times \text{Dom}(V)$*,*
- *l'ensemble d'états initiaux* $S^0 = \{\langle l_0, c \rangle \mid \Theta(c) = true \ \}$ *et* $S^0 \subseteq S$*,*
- *l'ensemble des actions* $\Lambda = \{\langle a, p_a \rangle \mid a \in \Sigma \wedge p_a \in \text{Dom}(\text{sig}(a))\}$ *partitionné dans les ensembles tels que pour* $\# \in \{?; !, \tau\}, \Lambda^\# = \{\langle a, p_a \rangle \mid a \in \Sigma^\# \wedge p_a \in \text{Dom}(\text{sig}(a))\}$*,*
- $\rightarrow$ *est la relation dans* $S \times \Lambda \times S$ *défini par :*

$$\frac{\begin{array}{c}\langle l, c \rangle, \langle l', c' \rangle \in S, \quad \langle a, p_a \rangle \in \Lambda, \\ t = \langle l^o, a, G, A, l^d \rangle \in T, \quad G(c, p_a) = true, \quad c' = A(c, p_a)\end{array}}{\langle l, c \rangle \overset{\langle a, p_a \rangle}{\rightarrow} \langle l', c' \rangle}.$$

La définition 6 indique le processus de création de la sémantique d'un IOSTS sous forme d'un IOLTS. Un état de l'IOLTS formé est un couple d'une localité et de l'affectation des variables V. Les états initiaux S^0 sont un sous-ensemble des états S, $S^0 \subseteq S$. Un état initial est un couple définit par la localité initiale et une affectation des variables satisfaisant les conditions initiales Θ. Une action $\lambda \in \Lambda$ de l'IOLTS est un couple défini par une action $a \in \Sigma$ de l'IOSTS et les valeurs des paramètres de l'action a. Ce couple forme l'étiquette de l'action λ. Enfin, les transitions sont étiquetées par les actions λ lorsque celles-ci peuvent être

tirées depuis l'automate IOSTS. Une transition peut être tirer si l'action a est possible depuis une localité l et si la valeur des variables du contexte et la valeur des paramètres de a satisfont la garde G. Elle aura comme effet de mettre à jour les valeurs des variables suivant la valeur des paramètres de a.

L'IOLTS résultant de la sémantique d'un IOSTS peut être un système de transition infinie en nombre d'états, mais aussi en nombre de transitions. Le nombre d'états dépend du domaine des variables. Si l'une d'entre elles est infinie, le nombre d'états de l'IOLTS sera infini. Le nombre de transitions en provenance d'un seul état peut lui aussi être infini. En effet, le nombre de transitions d'une action depuis un état est en relation avec le domaine des paramètres de cette action. Si le domaine d'au moins un des paramètres de l'action est infini, alors il y a une infinité de transitions dans l'automate IOLTS.

3.2.3 Opération sur les IOSTS

Cette section présente une partie des opérations portant sur les IOSTS. Les opérations sont des transformations d'un graphe d'un l'automate ou de plusieurs graphes d'automates.

Composition de deux IOSTS

La composition de deux IOSTS permet de synchroniser les actions d'entreé et de sortie des deux systèmes, mais n'influence pas les transitions étiquetées par des actions internes des deux systèmes. En d'autres mots, la composition permet de synchroniser le comportement visible de deux systèmes, mais ne bloque pas les systèmes dans leurs évolutions non observables. Cette opération impose que les deux systèmes soient compatibles [16]. C'est-à-dire qu'ils partagent le même alphabet visible, les mêmes paramètres et n'ai aucune variable commune.

Pour la suite de cette section, nous utiliserons les notations suivantes. Soit deux IOSTS $\mathcal{S}_\# = \langle D_\#, \Theta_\#, L_\#, l^0_\#, \Sigma_\#, T_\# \rangle$ pour $\# \in \{1, 2\}$ avec $D_\# = V_\# \cup P_\# \cup M_\#$ et $\Sigma_\# = \Sigma^?_\# \cup \Sigma^!_\# \cup \Sigma^\tau_\#$.

Definition 7 *$\mathcal{S}_1$ et $\mathcal{S}_2$ sont compatible pour une opération de composition si :*

- $V_1 \cap V_2 = \emptyset$ *- Les variables des systèmes sont uniques à un seul système, aucune variable n'est partagée.*
- $P_1 = P_2$ *- Les paramètres sont communs.*
- $\Sigma^!_1 = \Sigma^!_2$ *et* $\Sigma^?_1 = \Sigma^?_2$ *- L'alphabet visible est commun. Ce qui implique que toutes les actions visibles paramétrées ont la même signature dans les deux systèmes :* $\forall\ a_1 \in \Sigma^\#_1, a_2 \in \Sigma^\#_2 \mid a_1 = a_2\ :\ sig(a_1) = sig(a_2)$ *pour* $\# \in ?; !$

La composition est utilisée sur un système lors de la génération de tests (l'opération *produit* est alors utilisée) ou d'un test sur l'implémentation de ce système (l'opération *parallèle* est alors utilisée). Ces deux opérations sont une spécialisation de la composition générique.

Definition 8 *La composition générique de deux systèmes* $\mathcal{S} = \mathcal{S}_1|\mathcal{S}_2$ *entre deux IOSTS compatibles est un IOSTS* $\mathcal{S} = \langle V \cup P, \Theta, L, l^0, \Sigma, T\rangle$ *défini par :*

- $V = V_1 \cup V_2$ - *L'union des variables des deux systèmes.*
- $P = P_1 = P_2$ - *L'union des paramètres des deux systèmes.*
- $\Theta = \Theta_1 \wedge \Theta_2$ - *La conjonction des conditions initiales.*
- $L = L_1 \times L_2$ - *Une localité de* $\mathcal{S}$ *est le couple de deux localités de chacun des systèmes.*
- $l^0 = \langle l_1^0, l_2^0\rangle$ - *La localité initiale est le couple des localités initiales.*
- $\Sigma^\# = \Sigma_1^\# = \Sigma_2^\#$ *pour* $\# \in ?, !$ - *Les actions visibles sont les même pour chacun des systèmes.*
- $\Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$ - *Les actions internes sont l'union des actions internes de chacun des systèmes.*
- *L'ensemble des transitions* T *du système défini par les règles :*

$$\frac{\langle l_1, a, G_1, A_1, l_1'\rangle \in T_1, \quad a \in \Sigma_1^\tau, \quad l_2 \in L_2}{\langle\langle l_1, l_2\rangle, a, G_1, A_1 \bigcup_{x \in V_2} (x := x), \langle l_1', l_2\rangle\rangle \in T} \quad (3.1)$$

(de même pour $a \in \Sigma_2^\tau$)

$$\frac{\langle l_1, a, G_1, A_1, l_1'\rangle \in T_1, \ \langle l_2, a, G_2, A_2, l_2'\rangle \in T_2}{\langle\langle l_1, l_2\rangle, a, G_1 \wedge G_2, A_1 \cup A_2, \langle l_1', l_2\rangle\rangle \in T} \quad (3.2)$$

La règle 3.1 indique que le produit de deux IOSTS permet l'évolution indépendante des deux systèmes en fonction des transitions étiquetées avec des actions internes. Et la règle 3.1 indique que les actions observables sont synchronisées.

Les deux opérations de composition que sont le produit ($\times$) et la composition parallèle ($||$) sont présentées dans l'article [70].

Déterminisation

Un système défini par un IOSTS est déterministe si son IOLTS est déterministe [21]. Mais la déterminisation de cette l'IOLTS est généralement indéterminable. Comme il est impossible de toujours prouver qu'un système infini est déterministe, l'auteur de [52] introduit une notion plus forte qui est les systèmes de transitions symboliques (STS) structurellement déterministes. De plus, il prouve que des STS structurellement déterministes sont déterministes, d'où leur nom. Cette définition peut être transposée de manière triviale aux IOSTS.

Definition 9 *Un IOSTS* $\mathcal{S} = \langle D, \Theta, L, l_0, \Sigma, T \rangle$ *est structurellement déterministe si et seulement si*

$$\forall \langle l^o, a_1, G_1, A, l_1 \rangle \neq \langle l^o, a_2, G_2, A, l_2 \rangle \in T : (a_1 = a_2) \Rightarrow (G_1 \wedge G_2 = false)$$

et

$$\Sigma^\tau = \emptyset$$

Le comportement d'un IOSTS est déterministe si celui-ci présente des gardes mutuellement exclusives pour les transitions étiquetées par une même action et qu'il ne présente pas d'action inobservable (action interne). L'opération de déterminisation est décrite dans [51].

3.3 Contrôleurs

Un contrôleur est un mécanisme qui permet de restreindre le comportement d'un système. Imaginons un système disposant de capteurs et d'actionneurs. Tous les comportements possibles d'un tel système ne sont probablement pas ceux que nous souhaitons. Un comportement satisfaisant est atteint en fournissant des directives aux actionneurs en fonction de la vue de l'environnement donnée par les capteurs. Ces directives sont données par un système contrôleur. La figure 3.3 illustre le rôle du contrôleur vis-à-vis du système.

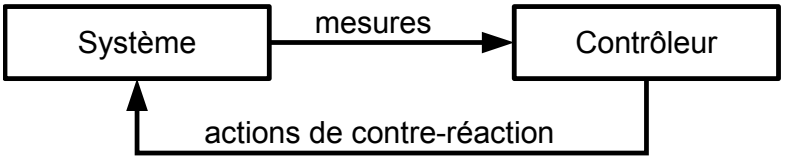


FIGURE 3.3 – Contrôleur de système

Le contrôleur et la synthèse de contrôleurs sont des aspects intéressants des automates IOSTS. En effet, à partir d'un ensemble de règles, un nouveau IOSTS est généré prenant en compte des règles définies par un contrôleur. De plus, les contrôleurs peuvent être construits à partir de propriétés recherchées. Pour résumer, ce mécanisme permet, à partir de propriétés souhaitées, de synthétiser un contrôleur et de créer l'automate IOSTS d'un système contrôlé. Par la suite, nous créerons un contrôleur et un système contrôlé différent afin de prendre en compte des règles d'une installation domotique.

3.3.1 Cadre général

Dans ce contexte, lorsqu'un système évolue, son contrôleur capte les informations de l'environnement en fonction des capteurs disponibles. Les évolutions ne peuvent pas toutes être mesurées, par manque de précision des capteurs par exemple. Sans compter les évolutions non observables, car il n'existe pas de capteur pour informer le contrôleur. C'est le problème d'*observation partielle*. Il ne faut pas confondre la non-observation du contrôleur avec les actions internes des IOSTS. La figure 3.4 illustre la vision du contrôleur d'un système. L'observation partielle peut être représentée comme un masque ou comme une projection de l'ensemble des états possibles sur un ensemble possiblement infini d'états observables. Plus d'information sur les observateurs et comment ils interfèrent avec les contrôleurs peuvent être trouver dans la thèse de Gabriel Kaylon [52].

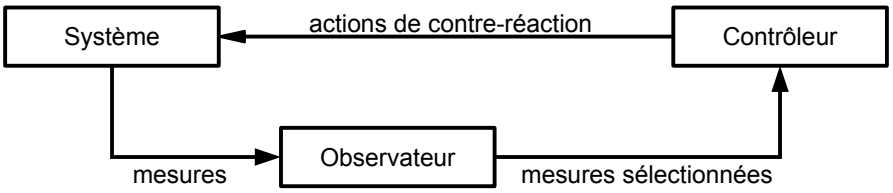


FIGURE 3.4 – Vue partielle d'un système par un contrôleur

L'observation partielle est une notion importante pour le cas du contrôleur, mais cette notion n'est pas présente dans le cas de la modélisation du comportement des dispositifs. Elle est présentée dans ce mémoire, car certaines formules utilisées par la suite sont définies dans la littérature en utilisant un observateur. Celles définies dans ce mémoire sont exemptes d'observateur. Le lecteur pourra se référer aux citations afin de prendre en compte l'observateur.

3.3.2 Définition formelle

Le but d'un contrôleur $\mathcal{C}$ est de contraindre l'évolution d'un système $\mathcal{S}$. Pour cela, le contrôleur observe le système et calcule depuis ses observations l'ensemble des actions contrôlables que le système ne doit pas exécuter pour s'assurer du comportement attendu du système. Le contrôleur peut interdire seulement les actions contrôlables du système, c'est à dire pour les IOSTS, les actions d'entreé. Il est défini comme un couple d'une fonction et d'un ensemble d'états initiaux [52].

Definition 10 *Soit un système* $\mathcal{S} = \langle V \cup P, \Theta, L, l^0, \Sigma, T \rangle$*, un contrôleur est une paire* $\mathcal{C} = \langle \mathcal{F}, E \rangle$ *définie par :*

- $\mathcal{F} : \langle L \times Dom(V) \rangle \rightarrow 2^{\Sigma}$ *est la fonction de supervision qui associe à chaque état* $\langle l, x \rangle \in \langle L \times Dom(V) \rangle$ *un ensemble* $\mathcal{F}(x)$ *d'actions contrôlables dont le déclenchement est interdit.*
- $E \subseteq Dom(V)$ *est l'ensemble des états qui restreint l'ensemble des états initiaux.*

Le modèle résultant de la construction d'un modèle commander $\mathcal{S}$ par un contrôleur $\mathcal{C}$ s'appelle le *système contrôlé*. Ce modèle est un IOSTS défini par [53] :

Definition 11 *Soit un système* $\mathcal{S} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ *et un contrôleur* $\mathcal{C} = \langle \mathcal{F}, E \rangle$*, le système* $\mathcal{S}$ *contrôlé par* $\mathcal{C}$ *est noté* $\mathcal{S}_{/\mathcal{C}} = \langle D, \Theta_{/\mathcal{C}}, L, l^0, \Sigma, T_{/\mathcal{C}} \rangle$ *tel que :*

- $\Theta_{/\mathcal{C}} = \Theta / E$ *- Les conditions initiales correspondent aux conditions initiales privées des états interdis* E*.*
- $T_{/\mathcal{C}}$ *est défini par la règle suivante :*

$$\frac{\forall \langle l^o, a, G, A, l^d \rangle \in T, \mid a \notin \mathcal{F}(l^o, v)}{\langle l^o, a, G, A, l^d \rangle \in T_{/\mathcal{C}}}$$

3.3.3 Synthèse de contrôleurs

L'objectif de la synthèse de contrôleurs est de créer un contrôleur $\mathcal{C}$ à partir de règles de comportement. Les contrôleurs peuvent être construits à partir d'un ensemble d'états qu'un système ne doit pas atteindre. Ces états sont déterminés comme inaccessibles soit parce qu'ils ne correspondent pas à une propriété souhaitée, soit parce qu'ils sont des états de blocage du système. Les états de blocage d'un système sont les interblocages ainsi que les blocages vivants. Les interblocages sont les états où un système ne peut plus être commandé, car il n'accepte aucune action. Les blocages vivants sont les états qui engendrent une boucle empêchant le système d'être commandé.

Le problème de la synthèse de contrôleurs est de calculer le contrôleur $\mathcal{C} = \langle \mathcal{F}, E, \rangle$ d'un système $\mathcal{S} = \langle D, \Theta, L, l^0, \Sigma, T \rangle$ tel que le système contrôlé $\mathcal{S}_{/\mathcal{C}} = \langle D_{/\mathcal{C}}, \Theta_{/\mathcal{C}}, L, l^0, \Sigma, T_{/\mathcal{C}} \rangle$ évite l'ensemble d'états Bad. Ce qui implique :

- $Reachable^{\mathcal{S}}_{T_{/\mathcal{C}}}((\Theta_0)_{/\mathcal{C}}) \neq \emptyset$
- $Reachable^{\mathcal{S}}_{T_{/\mathcal{C}}}((\Theta_0)_{/\mathcal{C}}) \cap Bad = \emptyset$

L'algorithme présenté de manière informelle pourrait être :

- Trouver les états $I(Bad)$ qui mènent aux états Bad par des transitions incontrôlables.

– Empêcher le système d'évoluer dans les états $I(Bad)$ en interdisant les transitions contrôlables.

Dans sa thèse, Kaylon [52] prouve que le calcul d'un tel contrôleur est indécidable par une méthode traditionnelle. Pour faire la synthèse d'un contrôleur, Kaylon utilise un mécanisme d'interprétation abstraite couplé de surapproximation [59]. L'interprétation abstraite indique que les calculs se réalisent sur les symboles et non sur les états. La surapproximation signifie que certains calculs donnent un résultat approché englobant la solution exacte afin de s'assurer de la terminaison de l'algorithme.

3.4 Synthèse

Ce chapitre introduit une extension des automates étendus, le modèle IOSTS. Dans le contexte de ce mémoire, les IOSTS permettent de modéliser un comportement de systèmes domotiques. Le choix de ce modèle est réalisé en fonction de la méthode de communication de ces systèmes et des besoins nécessaires à un point de commande.

La théorie des automates IOSTS est un outil mathématique permettant de modéliser des systèmes infinis au moyen de variables internes. Ces variables permettent de représenter les états de ce système. Nous avons vu le principe des transitions internes qui correspondent à un comportement d'un dispositif non observable. Ces transitions apportent un degré d'incertitude dans le suivi de l'évolution du système. Nous avons également présenté un opérateur qui permet de déterminiser les automates.

Pour finir, nous avons abordé, comment, au moyen d'un contrôleur, il est possible de contraindre le comportement d'un système. Les contrôleurs peuvent être générés à partir d'un ensemble d'états que l'on ne souhaite pas atteindre. Mais ce calcul est complexe est n'est pas sur de finir. Pour cela, nous avons évoqué des références qui utilisent un mécanisme de surapproximation, il permet de régler le problème d'indécidabilité de certains calculs.

II

Développement

Sommaire

Chapitre

4

Abstraction des objets communicants

En informatique, le concept d'abstraction intervient lorsque l'hétérogénéité matérielle ou logicielle devient complexe à gérer. L'abstraction est introduite à plusieurs niveaux, par exemple :

- Les systèmes d'exploitation permettent de se soustraire des préoccupations matérielles rencontrées pour la plupart des applications (accès à un périphérique de stockage, affichage...).
- Les langages de programmation permettent de s'affranchir du langage binaire de la machine. Ces langages proposent l'abstraction des structures de contrôle.
- Les langages de programmation de haut niveau affranchissent les programmeurs de la gestion de la mémoire nécessaire au stockage temporaire des éléments de structures de données.
- Les modèles sont une abstraction de différents aspects d'un problème. De nombreux modèles sont exprimés de manière graphique. Ils interviennent lors de la phase de construction d'un logiciel.

Dans notre cas, l'abstraction est nécessaire pour pouvoir prendre en compte plusieurs protocoles de communication de dispositifs et l'hétérogénéité des fonctionnalités des dispositifs. Elle permet de disposer d'un seul point d'entrée pour commander et obtenir des informations. Ce point d'entrée permet de créer des solutions génériques, c'est-à-dire des solutions qui ne sont spécifiques ni d'un média de communication ni des fonctionnalités des dispositifs.

En effet, les dispositifs peuvent être vus de différentes manières en fonction de l'utilité recherchée. Ainsi, un volet dispose de différentes fonctionnalités comme :

- D'un point de vue essentiel, un volet permet d'obstruer une ouverture.
- D'un point de vue de la sécurité des biens, un volet permet de renforcer la sécurité d'une fenêtre.
- D'un point de vue énergétique, un volet permet d'améliorer l'isolation thermique d'une ouverture...

4.1 Choix de l'ontologie

Dans la section 1.1 nous avons introduit l'abstraction des objets communicants comme un besoin pour les applications liées à la domotique. Nous avons vu dans la sous section 1.1.4 que : *le contexte est toutes les informations qui caractérisent une situation d'une entité*. Suivant cette idée, la prise en compte du contexte d'une installation domotique s'appuie en partie sur : les informations obtenues par les capteurs et les actions fournies par les actionneurs de cette installation.

L'abstraction des objets communicants est un sous-ensemble du contexte, c'est un aspect important des applications prenant en compte le contexte. Les applications utilisant uniquement les dispositifs d'une installation ont un choix plus important de méthodes pour mettre en oeuvre l'abstraction. Mais ces méthodes sont des solutions ad hoc, elles ne sont pas toutes efficaces pour la modélisation d'un contexte.

Les auteurs de l'article [78] présentent six méthodes de modélisation de contexte parmis lesquelles la modélisation clé-valeur, le modèle graphique, celle basée sur la logique, celle orientée objets ou encore, celle basée sur les ontologies. Les critères d'analyse sont :

1. La gestion de différents systèmes et de différents écosystèmes.
2. La vérification de la cohérence des informations constituant le contexte.
3. La gestion de la qualité des informations.
4. La gestion des ambiguïtés inhérentes à l'acquisition (par exemple l'unité d'acquisition, centimètre ou pouce).
5. Le niveau de formalisation du modèle.
6. L'utilisabilité de la méthode.

Parmi ces méthodes, ils expliquent que la modélisation basée sur un modèle objet et la modélisation basée sur les ontologies sont les meilleures approches avec une préférence pour les ontologies. Les autres méthodes ne satisfont pas le critère numéro quatre et au moins un autre critère parmi les cinq critères restants.

La programmation orientée objet est une abstraction des structures de données et des actions associées en tant qu'objets. Elle permet de modéliser les objets d'un problème donné et leurs interactions, et ainsi d'aider à la production d'un système capable de résoudre le problème. Comme le monde réel est caractérisé, lui aussi, par des objets et leurs interactions, une application développée en utilisant une approche orientée objet sera proche du problème. Les objets du monde réel sont des entités qui sont perçues de manière identique par de nombreuses personnes, par exemple : une matrice, la langue française, l'âme...

L'utilisation du paradigme de la programmation orienté objet nécessite la création d'un modèle objet. Ce modèle est intrinsèque à l'application réalisée, c'est-à-dire que le modèle est figé dans l'application. L'évolution du modèle implique une nouvelle version de l'application. Dans ces conditions, la prise en compte de l'évolution des technologies sera matérialisée par la livraison d'une nouvelle version de l'application. De plus, l'accès au modèle de l'application nécessite l'utilisation de techniques d'introspection lourdes à la mise en œuvre.

Compte tenu de ce qui précède, le modèle objet fait défaut dans le domaine :

- de la prise en compte de l'évolution des technologies liées à une application,
- de la possibilité d'explorer le modèle constitutif de l'application.

La modélisation basée sur des ontologies résout en partie ces problèmes. En effet, les ontologies permettent d'interroger les données (les objets) mais aussi le modèle (les concepts). Dans certaines conditions, la modification de l'ontologie est analogue à la modification de données dans une base de données ; la modification du modèle ne remet pas en cause l'application. L'opération de modification inclut le changement d'héritage ainsi que l'ajout ou la suppression de concepts et d'objets. Seuls quelques concepts piliers sont liés au code de l'application et leur modification nécessite la modification d'une partie du code de cette application, notamment les requêtes d'accès aux données de l'ontologie

Un objet correspond à un concept. Ils disposent tous deux d'une signification matérialisée par des relations avec d'autres concepts. Ainsi, l'utilisation des concepts et objets issus des ontologies est un moyen pour inclure un aspect sémantique dans une application. L'intérêt est de mettre à disposition des informations plus proches de la pensée humaine. Cette technologie consiste à rendre la connaissance manipulable par une machine. Pour autant, l'utilisation des ontologies ne permet pas à une machine de comprendre la connaissance, c'est le rôle de l'humain d'interpréter les informations délivrées par une application.

En somme, le choix d'utiliser des ontologies pour l'abstraction des dispositifs est dû aux qualités intrinsèques des ontologies que l'on ne retrouve pas dans les autres méthodes de modélisation :

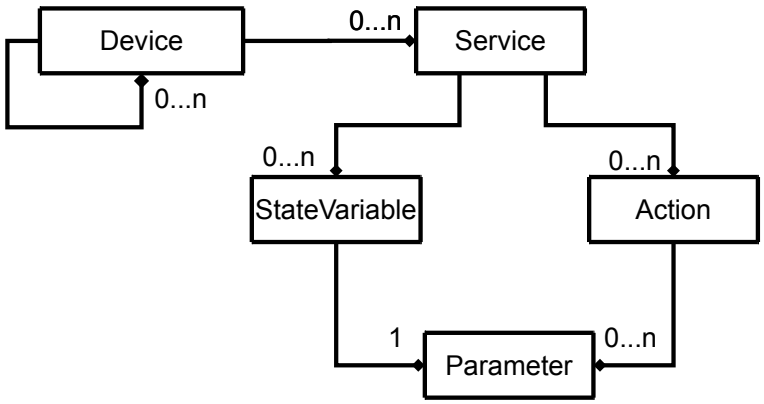


FIGURE 4.1 – Modèle du protocole UPnP (voir figure 4.2)

- la séparation des structures de contrôles et de la connaissance (c'est-à-dire du modèle de données et des données) ;
- la possibilité d'interroger le modèle de la même manière que les données ;
- l'élévation du niveau de compréhension des données ;
- la cohérence des données.

4.2 Modèle des objets communicants

Le protocole UPnP présente un modèle générique qui permet d'utiliser des fonctionnalités présentées sous forme de services. La figure 4.1 présente le modèle UPnP repris de l'article [20]. Les services UPnP sont une agrégation d'attributs (StateVariable) et de commandes (Action). Les profils publics disponibles auprès du Forum UPnP [1] permettent de décrire les services disponibles de quelques dispositifs standard. Lorsqu'un profil est manquant, une description spécialisée des services fournis par ce dispositif est mise en œuvre afin d'être accessible par les autres membres de l'écosystème UPnP. La description des services ne permet pas de savoir à quoi servent ces derniers. Pour cela, il faut assurer la diffusion de la compréhension des services du dispositif sur un autre média. Ce dernier problème implique une difficulté d'utiliser les dispositifs hors profil de manière automatique.

Le profil Home-Automation du protocole Zigbee possède un modèle intéressant pour la présentation des fonctionnalités des dispositifs et leur utilisation dans cet écosystème. La figure 4.2 représente une simplification du modèle Zigbee. Un

1. Site : http ://www.upnp.org ; dernier accès décembre 2011

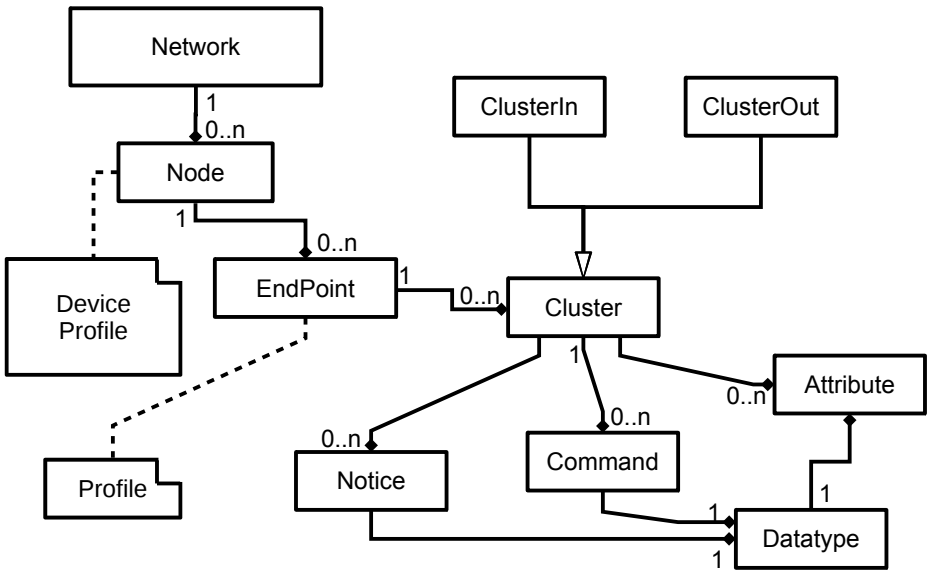


FIGURE 4.2 – Modèle du profil Home-Automation du protocole Zigbee

dispositif Zigbee (Node) dispose d'un ou plusieurs systèmes (ou End-Point). Un système est un élément pouvant fonctionner seul au sein d'un dispositif. Par exemple, si une prise multiple est un dispositif Zigbee, chacune des prises peut être commandée indépendamment. Chaque prise est ainsi un système de même type. Les systèmes sont décrits dans des spécifications de profils, par exemple le document [1] spécifie le profil *Home-Automation*. Chaque profil implémente des clusters, qui sont la description des fonctionnalités. Ces clusters ont pour rôle d'indiquer si la fonctionnalité est un point de commande ou un élément contrôlable et quels sont les attributs, les commandes et les notifications fournis par le système.

À l'instar de UPnP, il est possible de définir des profils qui permettent de créer de nouvelles fonctionnalités. Ces profils sont privés de même que les fonctionnalités qui leur sont associées. Les fonctionnalités privées ne peuvent pas être utilisées par les autres dispositifs sans qu'ils aient connaissance de ces profils privés. Ce problème a déjà été soulevé dans la première partie du mémoire, c'est le problème de la communication entre acteurs abordé à la section 2.3.

L'utilisation d'un modèle basé sur une ontologie permet de résoudre ce problème. L'idée est de s'inspirer de l'évolutivité du modèle UPnP ainsi que de la modélisation des dispositifs de Zigbee dans un modèle basé sur une ontologie qui permet de « comprendre » les informations.

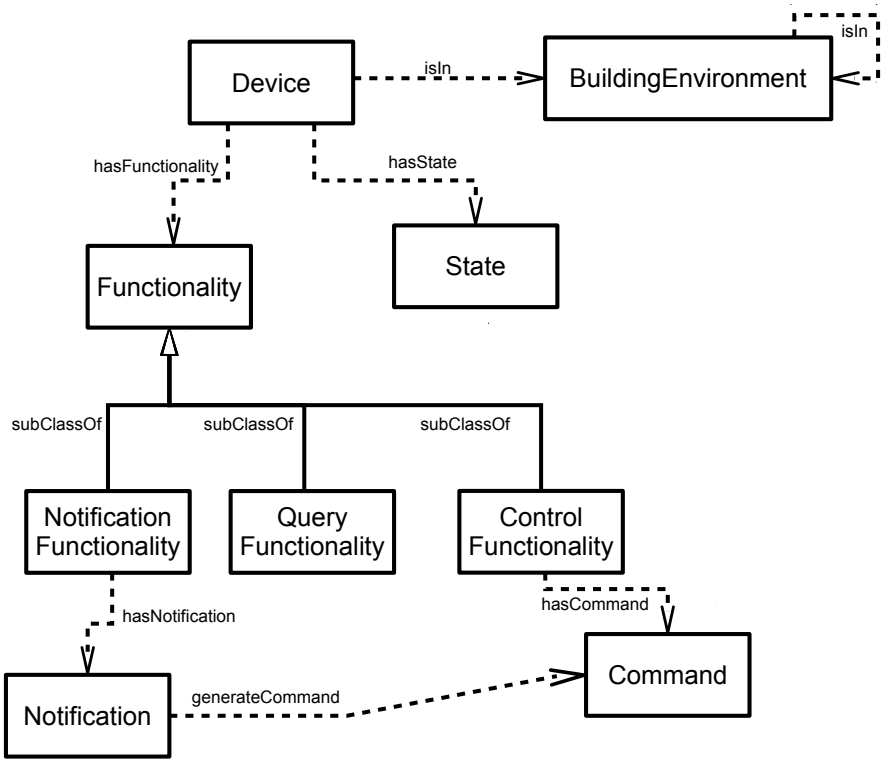


FIGURE 4.3 – Modèle qui permet l'abstraction des objets communicants

4.2.1 Éléments constituant les fonctionnalités de base d'un objet

Nous avons vu dans le chapitre 1 quelques projets développant des solutions domotiques plus ou moins génériques. Remarquons que la solution apportée par Dog présentée dans la sous-section 1.5.1 est la solution la plus spécifique des projets basés sur une ontologie. Dans le projet Dog, l'abstraction des objets communicants est réalisée par l'ontologie DogOnt [35]. Cette ontologie reprend des éléments clefs du modèle Zigbee.

Nous avons implémenté cette ontologie au sein du système d'information de l'entreprise Overkiz en tant qu'élément d'abstraction de dispositifs. Cette ontologie est fonctionnelle et répond bien aux attentes liées à la communication entre acteurs, à la multi-représentation et à l'évolution des technologies. En effet, nous avons implémenté dans cette ontologie de nouveaux protocoles de communication, de nouveaux dispositifs ainsi que de nouvelles fonctionnalités.

La figure 4.3 présente les différents concepts et leurs relations permettant de modéliser les objets communicants au moyen de l'ontologie DogOnt. Remarquons

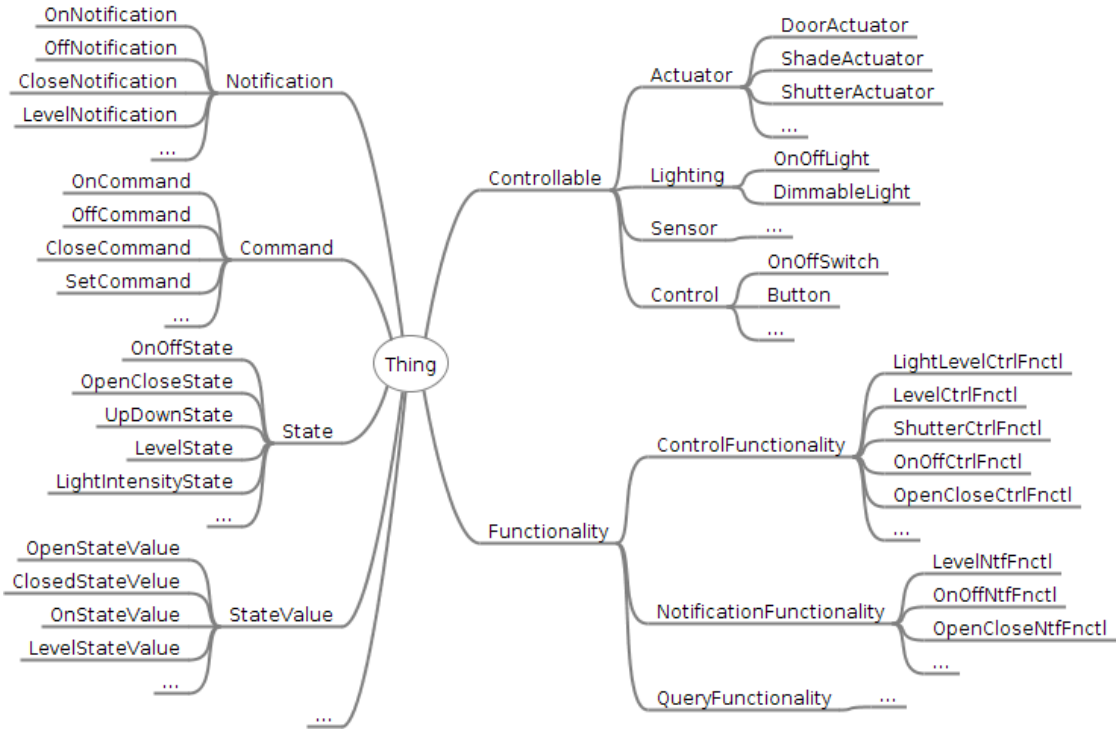


FIGURE 4.4 – Hiérarchie des concepts de l'ontologie

que le modèle de l'ontologie ne dispose pas du concept système comme nous l'avons vu dans le modèle Zigbee. La suite de cette section explique différents concepts rencontrés dans l'ontologie (en intension). De plus, un ou plusieurs exemples sont décrits par concepts (description en extension).

Il n'y a pas de lien entre les commandes et les états ; il est donc impossible de connaître l'effet de l'exécution d'une commande sur un dispositif et vice-versa. Le modèle ontologique ne permet pas :

- de savoir quelles sont les changements d'état d'un dispositif induits par l'émission d'une commande,
- de connaître les commandes qui permettent d'obtenir un état souhaité.

Ce manque est dû à la difficulté de modéliser le comportement des dispositifs. Ce problème est traité dans la suite de la thèse à partir du chapitre suivant.

La figure 4.4 présente un extrait de la classification d'héritage des concepts de l'ontologie. Elle indique quels sont les concepts enfants et parents des concepts décrits dans la figure 4.3. Les traits de la figure 4.4 représentent la relation *subClassOf* de l'ontologie, c'est-à-dire la relation d'héritage. Ainsi *Controllable* est un sous-concept de *Thing*.

FIGURE 4.5 – Prise multiple Zigbee

4.2.2 Le dispositif

Le concept « dispositif » modélisé par l'ontologie est ce qui correspond à un objet communicant vu du logiciel et pas forcement aux dispositifs tels que nous les percevons dans le monde réel. Le modèle ontologique ne possède pas de niveau « système » comme le concept *End-Point* du modèle Zigbee, ce qui implique l'impossibilité de modéliser des dispositifs composés de différents systèmes. Les dispositifs sont représentés en tant que nœuds fils du concept *Controllable* de l'arbre de la figure 4.4.

La prise multiple Zigbee[2] illustrée en figure 4.5 fournit trois emplacements délivrant de l'électricité et commandables individuellement par le protocole Zigbee. Pour chacun de ses emplacements, un système *End-Point* est capable de commander la distribution d'électricité et de surveiller la consommation du courant des éléments connectés.

L'absence du concept « système » dans l'ontologie a comme effet de perdre l'information de proximité géographique des systèmes. Une application utilisant l'ontologie telle qu'elle est décrite dans cette section, conceptualisera trois prises au lieu d'une prise composée de trois systèmes.

4.2.3 Les états

Un état d'un dispositif est une situation, une condition particulière de ce dispositif. Un état peut-être une valeur continue ou une information discrète. Par exemple, l'état ouvert d'un volet roulant (OpenStateValue) ou l'état fermé (ClosedStateValue) est une information discrète qui correspond à l'état d'ouverture (OpenCloseState). Son niveau d'ouverture (LevelState) en revanche, est une valeur continue (LevelStateValue). Cette valeur de l'état est représentée par une propriété du concept *LevelStateValue*.

2. Prise Zigbee du constructeur Netvox : http ://www.netvox.com.tw. (dernier accès en janvier 2012)

4.2.4 Les fonctionnalités

Une fonctionnalité d'un dispositif est une description sémantique des types d'actions réalisables. Elle permet de donner un sens aux structures de données qu'utilise un développeur. Ce dernier manipule, par exemple, le concept de fonctionnalité d'ouverture et de fermeture (*OpenCloseFnctl*) et non un identifiant de cluster comme c'est le cas dans Zigbee (identifiant 0x0100 pour le cluster *closure* [2]). Les fonctionnalités regroupent une ou plusieurs commandes ou notifications.

4.2.5 Les commandes et notifications

Les commandes sont des ordres à envoyer à des dispositifs d'un type donné afin que ces derniers effectuent une action dans le monde réel. Les notifications sont des informations provenant d'un dispositif. Elles informent d'une évolution de l'état d'un dispositif ou d'un changement provenant du monde réel. Les commandes et notifications ont un sens pour un concepteur de logiciel ou un utilisateur qui sait interpréter la sémantique des objets manipulés. Par exemple, la commande *OnCommand* de la fonctionnalité *OnOffCommandFnct* ordonne à un dispositif de se placer en état de fonctionnement. En revanche, une application manipule seulement la syntaxe des commandes et des notifications. En d'autres termes, l'application manipule des structures de données représentant des commandes ou notifications.

À l'instar des valeurs continues ou discrètes des états, les commandes et les notifications peuvent être paramétrées ou non. Lorsqu'elles ne sont pas paramétrées, les commandes et les notifications se suffisent à elles-mêmes. Par exemple, les commandes *OnCommand* et *CloseCommand* sont des commandes non paramétriques et le sens est trivial. En revanche, lorsqu'une commande ou une notification est paramétrée, la valeur du paramètre vient en complément de la sémantique de la commande. Par exemple, la commande *SetLevelCommand* est une commande paramétrique. Pour un volet roulant, elle indique à quel niveau d'ouverture le volet roulant doit se positionner.

Dans une application utilisant cette ontologie, il existe un composant logiciel par écosystème qui permet de transformer les concepts de l'ontologie en commandes réelles envoyées aux dispositifs et inversement, de transformer les notifications en provenance de dispositifs en concepts manipulables par l'application.

4.3 Évolution du modèle

Les ontologies sont remarquables par leur évolutivité (voir la section 2.3.2). Un modèle basé sur une ontologie peut être étendu sans influencer les composants logiciels n'ayant pas besoin du modèle étendu. Ainsi, nous pouvons décrire différentes fonctionnalités d'un objet tout en nous référant aux concepts utilisés par les composants déjà développés.

La section précédente décrit le modèle des fonctionnalités primaires d'un objet, c'est-à-dire les commandes qui permettent de faire fonctionner un dispositif et les notifications qui permettent de connaitre l'état d'un dispositif. En d'autres termes, les fonctionnalités primaires permettent de changer l'état d'un dispositif et de connaître son état. Les dispositifs produisent des fonctionnalités que nous appelons secondaires. Ces fonctionnalités secondaires sont des propriétés qui sont sous-entendues, elles ont une signification pour l'utilisateur.

Par exemple, nous pouvons « voir » les dispositifs comme étant des éléments de sécurité. Un volet fermé augmente la sécurité de l'installation et donc des personnes et des biens à l'intérieur de l'installation. Un autre exemple est l'aspect énergétique des dispositifs. Ces derniers se comportent comme des consommateurs d'énergie ou comme des créateurs d'énergie dans certaines conditions. Les volets ont aussi la faculté d'augmenter la résistance thermique des ouvertures et se comportent donc comme isolant lorsqu'ils sont fermés. Ouverts, les volets laisseront entrer la lumière lorsque l'environnement extérieur est lumineux.

La figure 4.6 décrit un modèle de gestion de l'énergie basé sur une ontologie. Ce modèle est une évolution du modèle présenté précédemment, mais il n'a pas été implémenté. En effet, nous nous sommes concentrés sur la modélisation du comportement.

Device et *State* sont des concepts présentés dans les figures 4.3 et 4.4. L'état d'un dispositif est important pour la gestion de l'énergie manipulée. Il permet de savoir si un dispositif consomme ou crée une certaine énergie. Un état peut être à la fois créateur d'une énergie et consommateur d'une autre. Par exemple, une lampe allumée consomme de l'électricité, mais crée de la lumière.

Les relations entre les concepts permettent de modéliser :

- Quelle est l'utilisation de l'énergie (consommer ou créer) d'une installation pour des conditions particulières ? Les conditions permettent de décrire quels sont les états de l'environnement externe ou interne qui conduisent à la création ou la consommation d'une énergie pour certains états de dispositifs.
- Quels sont les types d'énergie utilisés par l'état d'un dispositif ? Les énergies disponibles sont : l'électricité, la lumière, la chaleur, le froid.

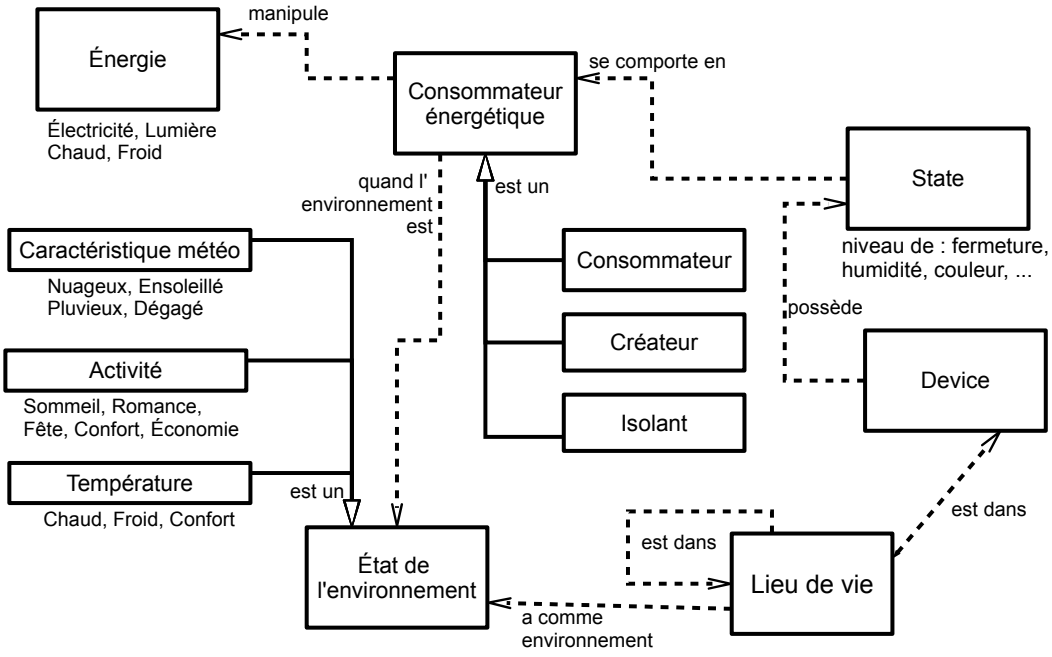


FIGURE 4.6 – Modèle de gestion de l'énergie

Remarquons que le froid n'est pas une énergie à proprement parler. Le concept *Énergie* présenté dans le modèle étendu ne correspond pas à la définition physique de ce terme. Il correspond plutôt à un élément immatériel pouvant être créé par un dispositif.

– Où se trouve le dispositif dans l'installation ? La description des lieux de vie est une autre préoccupation que la gestion énergétique des dispositifs et correspond à une autre partie du modèle. Voici quelques exemples de lieu de vie : jardin, appartement, maison, pièce, salle à manger, chambre... Notons qu'un lieu de vie peut-être inclus dans un autre lieu de vie ; par exemple une salle à manger est dans une maison. La localisation symbolique portée par cette partie du modèle permet de donner des informations de localisation géographique des dispositifs et de l'état de l'environnement. Il est alors possible de manipuler la lumière de la salle à manger.

Prenons un exemple concret : nous disposons de lampes et de volets commandables dans une pièce donnée. Nous souhaitons demander plus de lumière dans la pièce. Sachant que le temps dehors est ensoleillé, l'ontologie est interrogée pour connaître quels sont les états des dispositifs qui permettent de créer de la lumière. La requête est créée de manière à donner en priorité les états des dispositifs qui permettent de créer de la lumière sans consommer de l'électricité. De plus, cette

requête filtre les dispositifs localisés dans la pièce considérée. On obtient une liste d'états à atteindre afin d'augmenter la luminosité de la pièce.

La gestion de l'énergie telle que nous l'avons modélisée implique l'utilisation des états des dispositifs. Nous l'avons vu, le modèle basé sur une ontologie ne permet pas de connaître les commandes à envoyer pour obtenir un état. Pour résoudre ce problème, il faut modéliser le comportement des dispositifs. Un comportement forme un graphe où les états sont les nœuds et les arcs entre les états sont les commandes. Le chapitre suivant présente ce type de modélisation.

4.4 Synthèse

Dans ce chapitre, nous avons abordé l'abstraction des objets communicants au moyen d'un modèle basé sur l'ontologie DogOnt. Cette ontologie prend en compte différents aspects d'une installation et notamment la description des objets communicants. L'implantation de l'otologie au sein de l'entreprise Overkiz a confirmé que les ontologies sont un bon outil pour la gestion du contexte et que DogOnt est une ontologie de choix pour l'abstraction des objets communicants.

L'ontologie DogOnt reprend les principes du modèle des dispositifs et des fonctionnalités de Zigbee. L'utilisation des ontologies permet d'obtenir l'évolutivité du modèle UPnP en plus de la connaissance des fonctionnalités des dispositifs. En effet, cette technologie permet de manipuler de manière logicielle des structures de données portant un sens pour un utilisateur.

L'utilisation des ontologies permet de séparer les préoccupations de gestion du modèle de l'aspect traitement des informations. Par exemple, nous avons modélisé un prototype de modèle de la gestion énergétique des dispositifs sans remettre en cause les composants applicatifs se basant sur le modèle principal.

Nous avons établi que l'utilisation de ces deux modèles permet de connaître des informations sur les états des dispositifs en fonction de requêtes. Par contre, il n'est pas possible de connaître les commandes à envoyer pour obtenir les états souhaités. C'est pour cette raison que, par la suite, nous avons travaillé la modélisation du comportement des dispositifs.

Sommaire

Chapitre

5

Modélisation du comportement des objets communicants

La modélisation du contexte d'une installation domotique au moyen d'une ontologie ne permet pas à elle seule de gérer le comportement des dispositifs occupant l'installation. En effet, les ontologies ne disposent pas d'un cadre formel permettant de décrire les évolutions possibles des systèmes. Afin de modéliser le comportement des dispositifs, nous proposons d'utiliser les automates étendus de type IOSTS. Ces automates doivent rendre compte du comportement unitaire des dispositifs. L'avantage d'utiliser ce type de formalisation et de pouvoir réutiliser les travaux liés à la vérification des modèles et aux tests des systèmes.

L'utilisation des variables des automates IOSTS permet de décrire exactement l'état du dispositif. Les localités quant à elles, forment un ensemble fini. Les différentes valeurs des variables des localités permettent de représenter un ensemble d'états pour un dispositif. Ces valeurs peuvent représenter un terme qui est porteur de sens pour un utilisateur. Par exemple, un volet est soit fermé soit ouvert, ces deux termes sont ses localités. Le domaine de la variable des localités peut être $\{ouvert, fermé\}$. Lorsqu'il est ouvert, il possède une infinité d'états correspondant à son niveau d'ouverture, la localité ouverte est alors un ensemble infini d'états. Cet ensemble d'états d'ouverture peut être représenté par une variable $ouverture \in]0, 1]$ correspondant au pourcentage d'ouverture. Lorsqu'il est fermé, le volet est dans un seul état, la localité fermée est l'ensemble composé d'un seul est unique état fermé $ouverture = 0\%$.

Dans le cadre de ces travaux, nous nous intéressons uniquement aux systèmes qui sont en mesure de donner des informations relatives à leur état courant et à l'évolution de celui-ci. L'évolution d'un dispositif intervient suite à des stimuli par l'émission de notifications à destination de points de commande. Ces notifications sont des réactions engendrées en réponse à un changement d'état de l'environnement extérieur pour un capteur ou à un changement d'état pour un système. Le modèle IOSTS ne modélise pas les informations temporelles d'évolution entre un stimulus et la ou les réactions. De même, il n'existe aucune information qui permet de connaître le temps de réaction d'un dispositif. De manière informelle, la réaction d'un système est matérialisée par l'envoi d'une « photographie » de l'état du système à un instant donné.

Lorsque l'écosystème ne dispose pas du retour d'état des dispositifs, c'est le module de communication de l'écosystème de créer les notifications permettant de la gestion de l'évolution des dispositifs. Pour ce faire, ce module peut soit :

- Prendre en compte un temps d'évolution des dispositifs avant d'envoyer une notification. C'est le cas du protocole de communication X10.
- Aller chercher l'information de mise à jour auprès du dispositif. C'est le cas de l'écosystème Zigbee.

Le système est modélisé du point de vue d'un point de commande. Il correspond à la fois au comportement d'un dispositif et au comportement du medium de communication. Lorsqu'un point de commande reçoit effectivement une notification, c'est que cette notification a été émise par un dispositif et que le medium de communication a pu transmettre l'information. Lorsqu'un point de commande émet une commande à destination d'un dispositif, le medium de communication doit transmettre la commande jusqu'au dispositif, ce dernier doit réagir et retourner une notification. Mais le medium peut ne pas transmettre la commande ou la notification. Dans ce cas, le dispositif est dans un état inconnu.

5.1 Contraintes de commandes

Il existe trois problèmes liés à la commandabilité :

- la perte de message, ce problème est souvent pris en charge par le protocole de communication ;
- la multitude des points de commande : l'environnement possède plusieurs points de commande qui peuvent interagir avec un dispositif ;
- l'impossibilité physique de réaliser une action.

Un contrôleur doit traiter une commande en tenant compte de la possibilité qu'une commande envoyée ne soit pas prise en compte. En effet, cette possibilité

arrive dans le cas de la perte d'un message ou dans le cas où de multiples points de commande commandant un dispositif. Nous modélisons cette propriété par l'utilisation d'actions internes. Les actions internes permettent aussi de gérer les cas d'erreurs en indiquant quel est le chemin à prendre lorsque des erreurs surviennent.

5.1.1 Medium de communication

Les dispositifs domotiques sont commandés au moyen d'un medium de communication. Quel que soit ce medium, les messages échangés sont soumis à des perturbations qui peuvent les détériorer ou engendrer leur perte. Les protocoles de communication sont sécurisés face à la perte de message par des systèmes d'acquittement, mais disposent d'un nombre de répétitions limité afin de ne pas encombrer le medium de communication. Lorsqu'un message est perdu, il est impossible de connaître l'état du dispositif.

Le medium de communication peut être vus comme un système faisant correspondre deux systèmes indépendants :

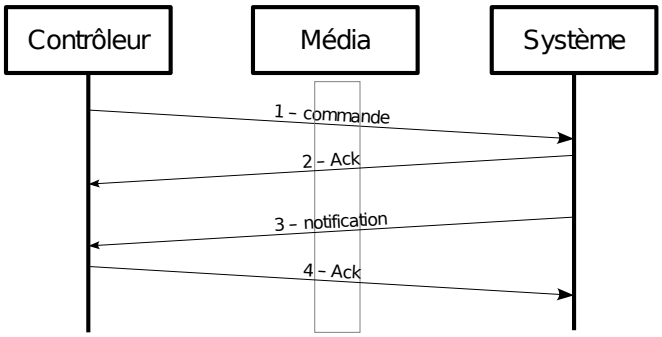
- $\mathcal{C} = \langle D_{\mathcal{C}}, \Theta_{\mathcal{C}}, L_{\mathcal{C}}, l^0_{\mathcal{C}}, \Sigma_{\mathcal{C}}, T_{\mathcal{C}} \rangle$ est le point de commande,
- $\mathcal{S} = \langle D_{\mathcal{S}}, \Theta_{\mathcal{S}}, L_{\mathcal{S}}, l^0_{\mathcal{S}}, \Sigma_{\mathcal{S}}, T_{\mathcal{S}} \rangle$ est le système domotique.

Ces deux systèmes disposent du même alphabet visible, mais les actions d'entrée de l'un sont les actions de sortie de l'autre : $\Sigma^?_{\mathcal{C}} = \Sigma^!_{\mathcal{S}}$ et $\Sigma^?_{\mathcal{S}} = \Sigma^!_{\mathcal{C}}$. Le medium de communication fait le lien entre l'envoi d'une commande par le point de commande et sa réception par le dispositif, c'est-à-dire entre l'alphabet de sortie du point de commande $\Sigma^!_{\mathcal{C}}$ et l'alphabet d'entrée du système domotique $\Sigma^?_{\mathcal{S}}$. De même pour les notifications, il fait correspondre $\Sigma^?_{\mathcal{C}}$ avec $\Sigma^!_{\mathcal{S}}$.

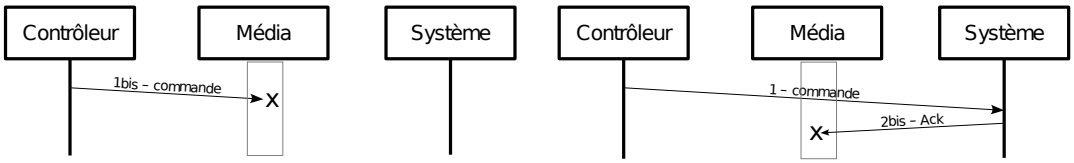
Nous souhaitons modéliser les dispositifs domotiques du point de vue d'un système de commande. Le système commandé prend en compte le medium de communication, la multitude de points de commande disponibles dans l'environnement ainsi que le dispositif domotique. La communication bidirectionnelle entre systèmes correspond en général au schéma présenté en figure 5.1.

La figure 5.1 donne le principe de communication entre deux systèmes. L'orientation des flèches indique le sens de la transmission, l'émetteur du message est à l'origine de la flèche et le récepteur à la fin. Le numéro des flèches donne l'ordre d'émission des messages et l'étiquette indique quel est le contenu du message. Les croix indiquent la perte d'un message.

Certains protocoles de communication mettent en place des mécanismes de répétition des messages perdus, mais le nombre de répétitions est limité. Ce mécanisme améliore la robustesse des échanges, mais ne les rend pas infaillibles. De plus, peu de protocoles de communication pour les systèmes embarqués possèdent



(a) Communication correcte entre deux systèmes



(b) Problème d'émission de commande

(c) Problème de réception de l'accusé de réception

FIGURE 5.1 – Communication entre deux systèmes

des mécanismes qui permettent de diffuser l'évolution des systèmes. La plupart du temps, il faut modifier le micrologiciel du récepteur afin d'écouter les ordres qui passent dans le médium de communication. Ces modifications ne font pas partie du protocole et sont sujettes à des bogues. Par conséquent, des messages peuvent être perdus comme le montre la flèche 2bis de la figure 5.1(c). La flèche 1bis de la figure 5.1(b) indique la perte de message pour les protocoles de communication sans mécanisme d'acquittement ; ainsi lorsqu'un point de commande envoie un message, il n'est pas certain que le dispositif le reçoive.

Lorsque les systèmes de contrôle doivent prendre en compte la perte de messages, ils connaissent l'état réel des dispositifs seulement lorsqu'ils reçoivent un message de la part de ces dispositifs, ce qui implique une non-connaissance de l'état des dispositifs le reste du temps. Ainsi, les modèles des dispositifs doivent disposer de la propriété suivante : à partir de tous les états d'un dispositif, il doit exister des transitions internes avec des actions de sortie menant à tous les autres états du dispositif, et ce, même si on ne sait pas quelle est la cause physique de ce changement d'état.

Dans la suite de ce mémoire, nous utilisons seulement des protocoles de communication avec acquittement. Ce qui implique la connaissance de la perte du message représenté par la flèche 2bis.

Dans le reste du mémoire, nous faisons l'hypothèse que les messages sont tou-

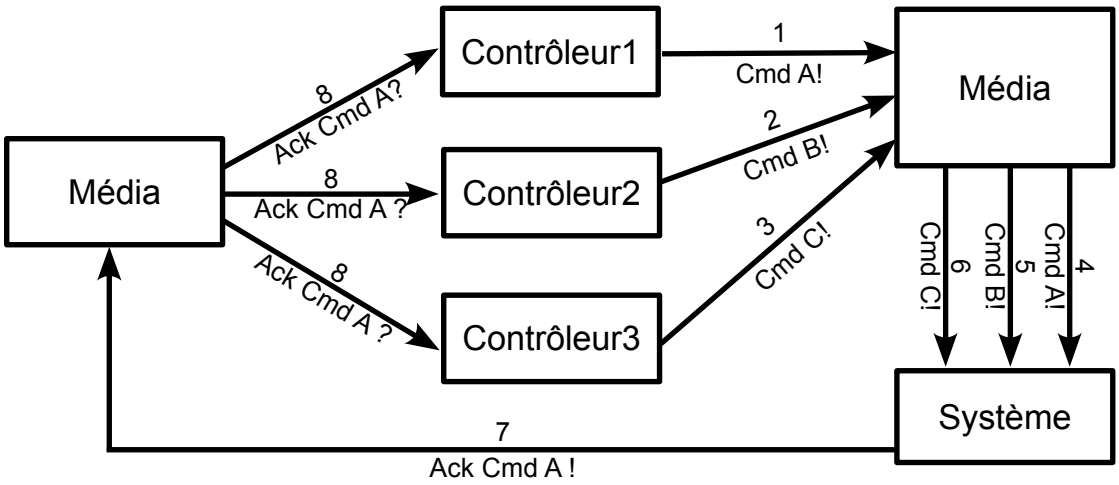


FIGURE 5.2 – Plusieurs points de commande pour un dispositif

jours transmis.

5.1.2 Multiples points de commande

Un autre aspect à prendre en compte est la multitude de points de commande disponible dans l'environnement d'un dispositif. En effet, l'environnement ne possède pas de contrainte sur le nombre de points de commande. Lorsqu'il y a plusieurs points de commande qui commandent un dispositif en même temps, ce dispositif exécute un seul des ordres. Ainsi lorsque les points de commande reçoivent la notification de retour, cette dernière ne correspond pas toujours à l'état souhaité par les points de commande. Il s'agit de modéliser un mécanisme qui prend en compte le caractère indéterminé des réponses reçues.

La figure 5.2 résume ce problème de multiples points de commande dans l'environnement. Elle présente trois points de commande envoyant simultanément trois ordres différents à un dispositif (les flèches 1 à 6). Le dispositif exécute le premier ordre et envoie une notification comme quoi il a bien exécuté l'ordre (flèche 7). Le medium de communication transmet l'information aux points de commande (flèches 8). Le dispositif est dans un état où il est incapable d'exécuter les ordres suivants (commande B et C), les contrôleurs reçoivent seulement la notification de la commande A. Les points de commande symbolisés par « Contrôleur 2 » et « Contrôleur 3 » s'attendent à recevoir respectivement une notification pour la commande B et C. Or, ces derniers reçoivent seulement l'acquittement pour la commande A.

Un exemple de ce cas pourrait être une télévision qui n'accepte aucun autre

message que la commande « ON » lorsque celle-ci est en veille. Cette commande correspond à la sortie de veille du dispositif afin de le rendre actif et fonctionnel. La commande A de la figure 5.2 demande la mise en veille de la télévision et les deux autres commandes ordonnent un changement de chaîne. Lors de la réception du message 4, la télévision s'éteint et provoque ainsi la non-exécution des ordres suivants. Ainsi, les contrôleurs 2 et 3 ne reçoivent pas le message de retour attendu.

5.1.3 Prise en compte des contraintes

Du point de vue d'un système, la réception d'une action réalisable crée un changement d'état jusqu'à obtenir l'état souhaité. Cette évolution peut être continue dans le cas d'un volet par exemple, ou discrète dans le cas d'une lampe. La modélisation de l'activité du système fait intervenir deux temps :

- un temps d'attente où le dispositif reste dans un état et attend une commande,
- un temps d'activité, où le dispositif traverse une multitude d'états.

Du point de vue du contrôleur, l'émission d'une action réalisable engendre une attente de la réalisation de l'action. La réalisation de l'action est matérialisée par la réception d'une notification du dispositif. La modélisation de l'activité d'un système pour le contrôleur fait intervenir deux temps :

- Un temps où le dispositif est dans un *état stable*, c'est-à-dire que le dispositif n'évolue pas et reste dans l'état où il est.
- Un temps où une commande est envoyée et où le contrôleur attend une réponse. Dans ce cas, le dispositif est dans un état non connu, il est soit en activité pour réaliser la commande envoyée, soit en erreur, soit en activité suite à une action d'un autre contrôleur... Nous appelons ce temps un *état virtuel* car cela ne correspond pas à un état réel du système.

Nous souhaitons prendre en compte le dispositif du point de vue du contrôleur, c'est-à-dire ne pas prendre en compte l'aspect continu de l'évolution du dispositif. Ainsi, nous avons besoin de modéliser les *états stables*, l'évolution du dispositif lorsqu'un autre point de commande commande le dispositif et les messages qui permettent de commander le dispositif. Les états stables sont représentés par les localités du modèle IOSTS et correspondent à des localités stables. L'évolution du dispositif correspond aux transitions entre les localités stables et ces transitions sont toujours étiquetées par des actions de sortie ($a \in \Sigma^!$).

Nous modélisons du point de vue du contrôleur, mais nous modélisons le dispositif. C'est-à-dire que :

- les commandes sont reçues par le dispositif et appartiennent à l'alphabet des actions d'entrée,

– les notifications sont envoyées depuis le dispositif et appartiennent à l'alphabet des actions de sortie.

La modélisation des messages de commande, étiquetés par des actions d'entrée ($a \in \Sigma^?$), est représentée par des transitions provenant de localités stables vers des localités que nous appelons virtuelles. Une localité virtuelle correspond à un état virtuel indiquant l'évolution potentielle d'un dispositif et l'attente d'un retour. Lorsque le contrôleur reçoit un retour du dispositif, si ce retour est celui qu'il attend, alors le dispositif est dans l'état stable souhaité. Si ce retour n'est pas celui attendu, alors le dispositif n'est pas dans l'état stable souhaité.

Lorsque le retour n'est pas celui attendu, le contrôleur a besoin de savoir quelle est l'action réalisée par le dispositif. Pour cela, nous modélisons le retour vers l'état précédent par une action interne. Cette action interne doit récupérer le contexte de l'état stable précédent.

Ce comportement ne correspond pas à celui du dispositif. C'est un mécanisme de gestion d'erreur qui est mis en place pour le contrôleur. Par la suite, le contrôleur gère la commande de retour à partir de l'état stable précédent.

Cette section présente les problèmes de modélisation des commandes, des notifications et des multiples points de commande. Elle décrit textuellement comment sont pris en compte ces problèmes. La section suivante présente de manière formelle leur prise en compte.

5.2 Particularités des IOSTS appliqués aux objets communicants

Afin de modéliser le comportement des dispositifs domotiques, nous introduisons les automates de comportement (AC). Ces automates utilisent le formalisme des IOSTS présenté dans le chapitre 3. Notons que les dispositifs domotiques sont modélisés du point de vue d'un système de contrôle et non du point de vue du dispositif. Cette remarque est importante, car ce modèle prend en compte le type de communication asynchrone, c'est-à-dire que la réaction à une sollicitation est décalée dans le temps. En effet, les dispositifs domotiques peuvent être commandés depuis de multiples points de commande (dans le cas où il existe plusieurs télécommandes). C'est pourquoi un contrôleur ne peut être sûr qu'une commande a été exécutée que lorsqu'il reçoit la notification de retour correspondante.

Dans un premier temps, nous abordons la signification de la notion de localité pour les dispositifs domotiques. Ensuite, nous notons comment sont modélisées les actions, puis nous étudions en détail quels sont les mécanismes qui permettent de modéliser le comportement des dispositifs du point de vue d'un contrôleur.

5.2.1 Localités des IOSTS et états des ontologies

Une localité définit un ensemble d'états. En effet, chaque localité $l \in L$ d'un IOSTS contraint indirectement le domaine des variables de l'automate en fonction des transitions ayant pour destination la localité l.

Les localités du modèle IOSTS sont les états des dispositifs tels qu'ils sont modélisés par l'ontologie (représenté par le terme State). Pour mémoire, le modèle ontologique est représenté par la figure 4.3 page 70. Les localités permettent de nommer un ensemble d'états d'un dispositif et donner ainsi une signification précise à des acceptions utilisateur.

Par exemple, un volet peut être fermé ou ouvert. La localité fermée correspond à l'état fermé du volet, alors que la localité ouverte correspond à l'état non fermé du volet. C'est-à-dire, si v est la variable correspondant au taux d'ouverture du dispositif en pourcentage :

- la localité fermée contraint la variable v au domaine $\{0\}$,
- la localité ouverte contraint la variable v au domaine $]0, 100]$.

5.2.2 Lien entre les actions et les localités

Les notifications sont matérialisées par les actions de sortie du modèle IOSTS. Elles forment l'ensemble des actions de sortie $\Sigma^!$. Une action de sortie $a \in \Sigma^!$ signifie qu'un dispositif envoie cette notification à un moment donné. Un dispositif émet une notification pour indiquer un changement de son état.

Les actions d'entrée correspondent aux commandes des dispositifs modélisées par l'ontologie. Elles forment l'ensemble des actions d'entrée $\Sigma^?$. Une action d'entrée $a \in \Sigma^?$ signifie qu'un dispositif est en mesure de recevoir à un moment donné cette commande. Lorsqu'un dispositif domotique accepte une commande, il réagit et donne une notification en retour.

Lorsqu'un contrôleur émet une commande, il attend la réponse d'un dispositif. Le contrôleur ne connaît pas l'état réel du dispositif, ce dernier peut être en train de réaliser la commande ou non. Afin de modéliser la non connaissance de l'état du dispositif nous fractionnons l'ensemble des localités en deux sous-ensembles $L = \underline{L} \cup \overline{L}^v$ tels que :

- $\underline{L}$ est le sous-ensemble des localités stables correspondant aux états réels du dispositif,
- $\overline{L}^v$ est le sous-ensemble des localités virtuelles correspondant aux états où le contrôleur attend une réponse d'un dispositif. L'état réel du dispositif n'étant pas connu du contrôleur.

Les transitions étiquetées par les actions d'entrée, de sortie et les actions internes sont appelées respectivement transitions d'entrée, de sortie et transitions internes. Pour faciliter l'écriture, nous indiçons les transitions avec l'indice correspondant à celui de l'alphabet de l'action de l'étiquette : $a \in t^{\#}$ ssi $a \in \Sigma^{\#}$ avec $\# \in \{?, !, \tau\}$.

Une transition d'entrée a toujours comme source une localité réelle et comme destination une localité virtuelle. Réciproquement, toute transition ayant pour source une localité réelle et destination une localité virtuelle, est une transition d'entrée. Ainsi, pour toute transition $t = \langle l^o, a, G, A, l^d \rangle \in T$, nous avons :

$$a \in \Sigma^{?} \iff l^0 \in \underline{L} \ \wedge \ l^d \in \overline{L}^{v}.$$

Une transition de sortie a toujours comme destination une localité réelle et toute transition à destination d'une localité réelle est une transition de sortie. La localité source peut être réelle dans le cas où la notification est reçue sans que le contrôleur n'ait émis une commande préalablement ou virtuelle dans le cas où le contrôleur ait envoyé une commande permettant la réception de la notification. Donc, pour toute transition $t = \langle l^o, a, G, A, l^d \rangle \in T$,

$$a \in \Sigma^{!} \iff l^d \in \underline{L}.$$

5.2.3 Définition des automates de comportement

Les automates de comportements permettent aux systèmes de contrôle de connaître le comportement des dispositifs d'une installation domotique. La connaissance de ces dispositifs permet aux systèmes de contrôle de calculer les chaînes de commandes pour aller d'un état à un autre état. Un automate de comportement (AC) est défini par un automate IOSTS auquel s'ajoutent des éléments sémantiques présentés par la définition 12.

Definition 12 *Un automate de comportement $\mathcal{S}_{AC}$ est un tuple $\langle D, \Theta, L_{AC}, L^0, \Sigma, T \rangle$ tel que :*

- *$L_{AC} = \underline{L_{AC}} \cup \overline{L_{AC}}^{v}$ est un ensemble fini non vide de localités et L^0 est l'ensemble des localités initiales $L^0_{AC} \subseteq \underline{L_{AC}}$ ($L^0_{AC} = \underline{L_{AC}}$ dans la plupart des cas).*
- *$T = T^{!} \cup T^{?} \cup T^{\tau}$ est un ensemble de transitions. Chaque transition de $T^{\#}$ est un tuple $\langle l^o, a, G, A, l^d \rangle$ tel que $a \in \Sigma^{\#}$ avec $\# \in \{?, !, \tau\}$.*

  De plus, $\forall \langle l, a, G, A, l' \rangle$ tel que $a \in T^{\tau}, G = \top, A = \emptyset, l \in \overline{L_{AC}}^{v}, l' \in \underline{L_{AC}}$.
- *$\underline{L_{AC}}$ est l'ensemble non vide de localités stables tel que :*

$$l \in \underline{L_{AC}} \iff \forall \langle l', a, G, A, l \rangle \in T, a \in \Sigma^{!} \cup \Sigma^{\tau}$$

– $\overline{L_{AC}}^{v}$ *est l'ensemble des localités virtuelles tel que :*

$$l^v \in \overline{L_{AC}}^{v} \iff \begin{cases} \exists!\langle l^o, a, G, A, l^v\rangle \in T, \ \textit{avec } a \in \Sigma^? \\ \exists!\langle l^v, a, \top, \emptyset, l^o\rangle \in T, \ \textit{avec } a \in \Sigma^\tau \\ \exists!\langle l^v, a, G', A', l^d\rangle \in T, \ \textit{avec } a \in \Sigma^! \end{cases}$$

Une trace d'une exécution ρ, notée $Trace(\rho)$, d'un automate de comportement $\mathcal{S}_{AC} = \langle D, \Theta, L_{AC}, L^0, \Sigma, T\rangle$ est une succession de localités et de transitions d'entrée ou de sortie notée $l_0\, t_0\, l_1\, \ldots\, t_{n-1}\, l_n$, mot du langage $L_{AC}.(T^{?!}.L_{AC})^*$. On a :

$l_0 \in L^0$, où $T^{?!} = T^? \cup T^!$
$\forall 0 \leq i \leq n-1, t_i = \langle l_i^o, a_i, G_i, A_i, l_i^d\rangle \in Trace(\mathcal{S}),\ |\ t_i \in T^{?!}\,,$ et $l_i^o = l_i\,, l_i^d = l_{i+1}$

5.2.4 Exemple

Cette section présente un exemple illustrant les concepts abordés dans ce chapitre. Comme nous l'avons vu dans le chapitre 4 et notamment à la section 4.3, un volet a les capacités d'augmenter l'isolation et d'obstruer une fenêtre. Dans cet exemple, le comportement d'utilisation (les fonctionnalités primaires) est le même que le comportement énergétique : (i) lorsque le volet est fermé, il obstrue la fenêtre et améliore l'isolation, (ii) lorsque le volet est ouvert, il dégage la fenêtre et ne change pas son isolation. Ces deux caractéristiques permettent de nommer les localités avec une vue tantôt utilisation tantôt énergie.

Le comportement des fonctionnalités secondaires d'un dispositif n'est pas toujours le même que le comportement d'utilisation. C'est le cas de la capacité d'un volet à laisser entrer la lumière dans une pièce. Ce comportement n'est pas compatible avec le comportement modélisant les fonctionnalités primaires. Dans ce cas, il faut créer un autre automate de comportement pour le volet. Par la suite, nous ne nous attardons pas sur les différentes « vues » des dispositifs. Nous utilisons seulement les fonctionnalités primaires.

Modèle ontologique

La figure 5.3 est une modélisation d'un type de volet disponible dans l'écosystème Zigbee. Ce type de volet est référencé dans le document de spécification du standard *Home Automation* de Zigbee [1, section *7.5.1 Shade*] page 61. Ce dispositif possède, par sa définition au sein du standard Zigbee, plus de fonctionnalités et de commandes que celles données par la figure 5.3, mais elles ne sont pas utiles pour illustrer notre exemple. L'utilisation de l'ontologie permet de séparer les connaissances du dispositif de leurs exploitations. Ainsi, la mise en place de

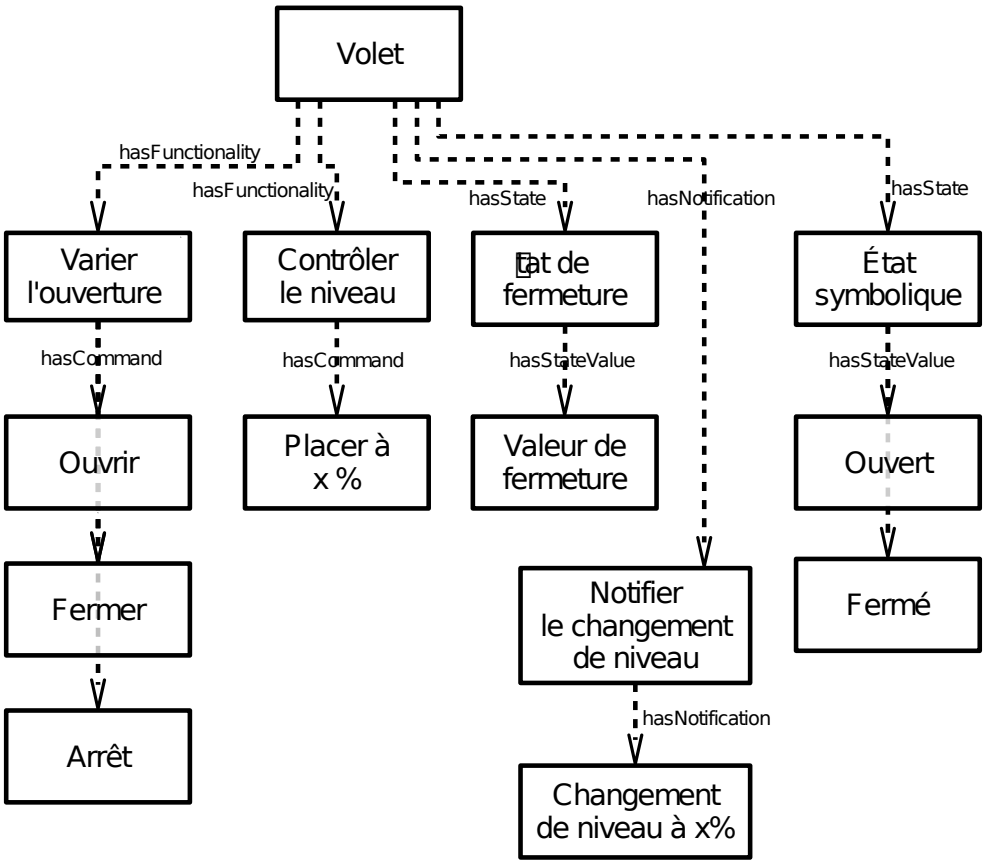


FIGURE 5.3 – Modélisation d'un volet Zigbee dans l'ontologie

ces fonctionnalités et commandes supplémentaires n'influence pas les explications fournies ici.

Le type de volet Zigbee possède deux fonctionnalités :

- La première, *Varier l'ouverture*, permet de positionner le volet dans un état souhaité. Le volet se meut en position ouverte (respectivement fermé) lorsque l'ordre donné est *Ouvrir* (*Fermer*). Le volet s'arrête de se déplacer lorsqu'il reçoit la commande *Arrêt* ou s'il arrive en butée d'ouverture (de fermeture).
- La seconde, *Contrôler le niveau*, permet de faire varier le taux d'obstruction de la fenêtre au moyen de la commande *Placer à*. Cette commande est paramétrée. Ce paramètre est exprimé dans l'ontologie par une fraction du niveau de fermeture alors qu'il est exprimé dans le protocole Zigbee par un entier sur 8 bits.

Les volets Zigbee ne possèdent pas de fonctionnalité de notification de fin d'ac-

tion. La préoccupation de s'assurer que le dispositif exécute l'ordre est déléguée à la partie communication de l'application. Cette dernière doit intégrer par exemple un système qui interroge le dispositif à intervalle de temps régulier et envoie une notification lorsque le dispositif a changé d'état. La fonctionnalité de notification est donc assurée par la partie communication de l'application. Elle permet de signaler le changement de niveau d'un volet par la notification *Changement de niveau à*.

Les commandes telles que « Varier l'ouverture » et « Contrôler le niveau » correspondent aux actions d'entrée du modèle IOSTS. La notification *Changement de niveau à* correspond à l'action de sortie du modèle IOSTS du volet.

Il existe deux de types de variables d'états pour le type de volet Zigbee :

- La variable *état de fermeture* correspond au taux de fermeture d'un volet. C'est une valeur continue qui est exprimée en fraction du taux d'obstruction de la fenêtre.
- La variable *état symbolique* du volet correspond à une énumération de symboles (des mots) représentant l'état du volet. Il est indiqué qu'un volet peut être dans l'état fermé, c'est-à-dire que la fenêtre dont il dépend est complètement obstruée. Lorsqu'il est ouvert, le volet n'est pas fermé. C'est-à-dire que le taux de fermeture n'est pas de 100%.

Les états modélisés par l'ontologie correspondent aux variables et parfois aux noms des localités des automates IOSTS. Dans le cas du volet, les termes permettant de nommer l'*état symbolique* du volet sont bien adaptés pour nommer les localités de l'IOSTS. En effet, celles-ci donnent une signification réelle aux localités. Quant à l'*état de fermeture*, il sera stocké dans une variable réelle bornée entre 0 et 100.

Comportement formel

La définition formelle de l'IOSTS du volet $\mathcal{S} = \langle D_{\mathcal{S}}, \Theta_{\mathcal{S}}, L_{\mathcal{S}}, L^0_{\mathcal{S}}, \Sigma_{\mathcal{S}}, T_{\mathcal{S}} \rangle$ est la suivante :

Definition 13

- $D_{\mathcal{S}} = V \cup P$, *avec* $P = \{x\}$ *où* x *est le paramètre de domaine* $[0\ ,\ 100]$ *des actions not et com et* $V = \{ferm, tmp\}$ *où* $ferm$ *est le niveau de fermeture du volet en pourcentage* ($Dom(ferm) = [0\ ,\ 100]$) *et tmp est une variable temporaire permettant de mémoriser la valeur du paramètre donné à l'action d'entrée précédente.*
- $\Theta_{\mathcal{S}} = true$, *il n'y a pas de condition initiale.*

- $L_{\mathcal{S}} = \underline{L} \cup \overline{L}$, *avec* $\underline{L}$ *composé de deux localités stables nommées par les termes des* états symboliques $\underline{L} = \{\text{“}ouvert\text{”}, \text{“}fermé\text{”}\}$ *et* $\overline{L}$ *composé de quatre localités virtuelles* $\overline{L} = \{S1, S2, S3, S4\}$.
- $L^0_{\mathcal{S}} = \underline{L}$, *toutes les localités stables sont des localités initiales.*
- $\Sigma_{\mathcal{S}} = \Sigma^! \cup \Sigma^? \cup \Sigma^\tau$.
  - *Avec les actions d'entrée du volet* $\Sigma^! = \{com, fermer, ouvrir\}$ *sachant que com est la commande paramétrée correspondant à* Placer à x% *de la figure 5.3, et fermer et ouvrir sont les commandes du même nom ;*
  - *l'action de sortie de notification* $\Sigma^? = \{not\}$ *;*
  - *l'action interne* $\Sigma^\tau = \{\tau\}$.
- $T_{\mathcal{S}} = \bigcup_{n=1}^{16}\{t_n\}$ *est un ensemble de 16 transitions, tel que chaque transition est un tuple* $t_n = \langle l^o_n, a_n, G_n, A_n, l^d_n\rangle$ *dont en voici une partie :*
  - $t_1 = \langle \text{“}ouvert\text{”}, not, (p_2 < 100), [ferm := p_2], \text{“}ouvert\text{”}\rangle$
  - $t_2 = \langle \text{“}ouvert\text{”}, com, (p_1 < 100), [tmp := p_1], S4\rangle$
  - $t_3 = \langle S4, \tau, true, [nop], \text{“}ouvert\text{”}\rangle$
  - $t_4 = \langle S4, not, (p_2 = tmp), [ferm := tmp], \text{“}ouvert\text{”}\rangle$
  - ...

Nous savons que la localité fermée est un état. Cet état correspond à la fermeture complète du volet. Formellement, cela signifie que $ferm = 100$ lorsque la localité courante est “$fermé$” et que $ferm \neq 100$ lorsque la localité courante est “$ouvert$”.

Les localités virtuelles correspondent à l'attente d'un retour du volet. Cela modélise le fait que le volet est en cours traitement de l'ordre et probablement en train de se mettre dans l'état commandé. Du point de vue du contrôleur, cet état correspond à une non connaissance de l'état du dispositif.

Représentation graphique

La figure 5.4 est une représentation graphique du type *volet Zigbee* présenté précédemment en définition 13. Dans ce type de représentation, les formes circulaires correspondent aux localités et les flèches sont les transitions entre les localités.

Les formes circulaires sont de deux types :

- Les ovales sont les localités stables de l'automate, elles permettent de modéliser les états du dispositif modélisé.
- Les cercles, plus petits que les ovales, sont les localités virtuelles du modèle.

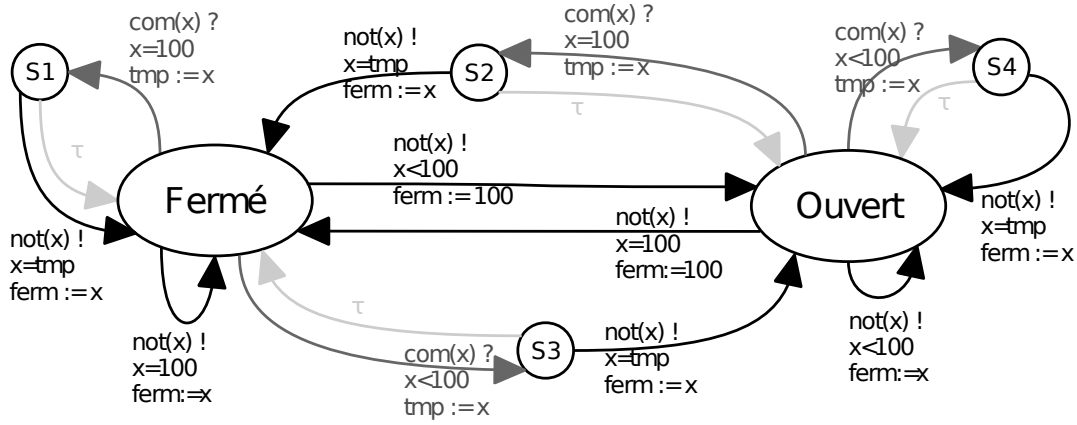


FIGURE 5.4 – Représentation graphique du type volet

Les transitions entre les localités sont représentées dans les graphiques par des flèches étiquetées avec du texte. Rappelons qu'une transition est définie par : $t_i = \langle l_i^o, a_i, G_i, A_i, l_i^d \rangle$. La source de la flèche est l_i^o, la destination est l_i^d. L'étiquette de la flèche se constitue d'au maximum 3 lignes :

- La première ligne est le nom de l'action suivi des noms des paramètres mis entre parenthèses ainsi qu'un symbole permettant de savoir à quel sous-ensemble de l'alphabet appartient l'action. Le symbole *!* signifie que l'action est une action de sortie ($\Sigma^!$) et le symbole *?* signifie que l'action est une action d'entrée ($\Sigma^?$). Le symbole τ, quant à lui indique que c'est une action de l'alphabet des actions internes Σ^τ.
- La deuxième ligne est présente lorsqu'il y a une condition différente de *true*. Elle indique quelle est la condition qui permet de déclencher la transition lors de l'apparition de l'action.
- La troisième ligne est présente lorsqu'il y a des affectations de variables.

Les différences de couleurs des flèches permettent d'améliorer la lisibilité du graphique. Les flèches les plus claires sont les transitions internes, les flèches gris foncé sont les transitions d'entrée et les flèches noires sont les transitions de sortie.

Nous remarquons sur la figure 5.4 que les localités virtuelles S1 à S4 présentent toutes le même schéma de transition. En effet, chacune d'elles est la destination d'une transition étiquetée par une action d'entrée et est la source d'une transition interne et d'une transition de sortie. De plus, la transition d'entrée dispose d'une affectation d'une variable temporaire et la transition de sortie compare le paramètre de l'action de sortie avec cette variable temporaire. Ce schéma peut donc être raccourci pour faciliter la lecture.

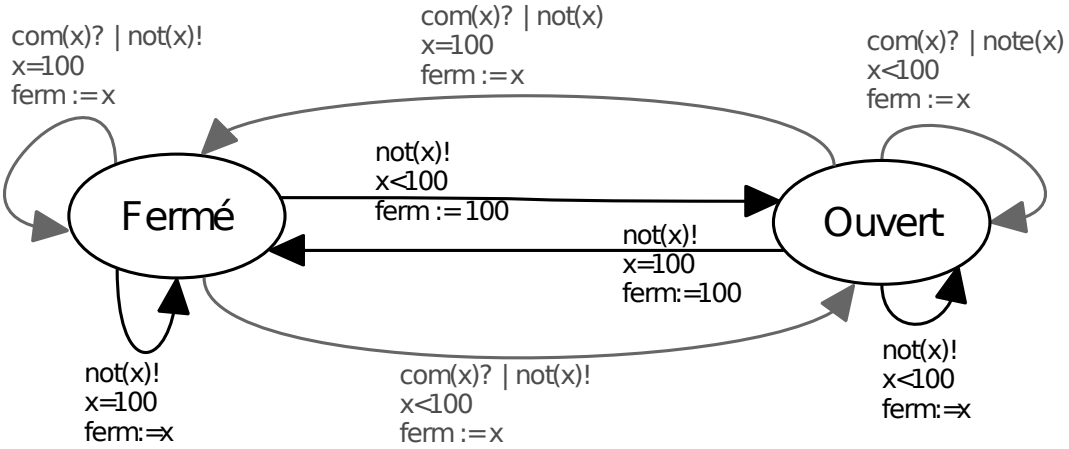


FIGURE 5.5 – Représentation graphique du type de volet avec raccourcie de notation

5.2.5 Raccourci de représentation

Comme nous l'avons vu précédemment, la représentation graphique d'une action d'entrée fait intervenir une localité virtuelle, une notification de retour et une action interne. Ce schéma est systématique, et il peut être abrégé par une notation indiquant l'action d'entrée et l'action de sortie. La représentation du volet avec le raccourci de représentation est illustrée par la figure 5.5.

La première ligne de l'étiquette d'un raccourci de notation indique la commande que doit recevoir un dispositif et la notification qu'il émet en retour. Lorsqu'il y a des paramètres, ils sont indiqués avec les mêmes symboles dans la commande que dans les notifications. Lors de l'écriture non raccourcie, ces symboles indiquent la présence de variables temporaires qui permettent de garder en mémoire les paramètres de commandes et les comparer avec les notifications de retour.

De même, les nuances de gris facilitent la lecture du graphique. Les flèches noires indiquent une transition de sortie. Les flèches grises indiquent un raccourci de notation, nous les nommons des transitions d'entrée-sortie.

5.3 Règles de comportements

La modélisation du comportement répond à la question : « Quelle est la notification et l'état de réponse d'un dispositif lors de la réception d'une commande ? » Un système possède des fonctionnalités, ces dernières permettent de réaliser des tâches particulières. Et afin d'utiliser ces fonctionnalités, un système doit être contrôlé en

lui envoyant les bonnes commandes permettant d'obtenir un comportement désiré. Ici l'objectif est de créer le contrôle de dispositifs.

Les automates IOSTS permettent de modéliser un comportement. À cet égard, nous avons vu comment modéliser le comportement d'un dispositif. La modélisation du comportement permet de connaître quelles sont les commandes à envoyer à un dispositif afin d'obtenir un état. Dans le même ordre d'idée, les IOSTS permettent de modéliser le comportement d'un ensemble de dispositifs. Ces dispositifs peuvent dépendre les uns des autres ou non.

- L'exemple d'un comportement de dispositifs ne dépendant pas les uns des autres pourrait être le comportement d'une chambre disposant d'une lampe de plafond, d'une lampe de chevet et d'un volet.
- L'exemple d'un comportement de dispositif dépendant les uns des autres pourrait être le comportement d'une télévision reliée à un décodeur satellite.

L'objectif est de créer un mécanisme qui permet le contrôle de dispositifs. L'ensemble des états possibles d'un sous-ensemble de dispositifs d'une installation domotique est le produit cartésien de tous les états possibles de ses dispositifs. Il n'est pas envisageable d'énumérer tous les états souhaités afin de définir le contrôleur de dispositifs. En revanche, il est possible de décrire un contrôleur au moyen de règles de réactions à des évènements ou des règles de contraintes. Les règles de réactions à des évènements consistent à exprimer une ou des actions à entreprendre lorsqu'un évènement apparaît et que le système est dans un certain état. Nous nommons ces règles : *Évènement Condition Changement* (ECC). Les règles de contraintes se présentent sous forme d'expressions booléennes.

Le comportement standard d'un système « dispositif – télécommande » est de réagir à chaque appui sur un bouton en effectuant une action. Dans ce cas, l'appui sur un bouton est un évènement traduit par la réception d'un signal pour le dispositif. La réaction réalisée par le dispositif lors de la réception d'un évènement est inscrite dans un programme. Dans un même ordre d'idées, les règles ECC permettent de programmer la réaction du système lors de la réception d'un évènement.

Les règles de contraintes permettent de matérialiser un ensemble d'états autorisés ou interdits d'un système.

À partir de ces deux types de règles, un contrôleur sera synthétisé afin de limiter le comportement du système.

5.3.1 Comportement sur évènements

Le comportement sur évènements est défini par des règles *Évènement Condition Changement* (ECC). Les ECC sont utilisées pour commander un système $\mathcal{S}2$ depuis

un autre $\mathcal{S}1$. Cette approche est flexible et permet à un système de commander un ou plusieurs autres systèmes, mais aussi d'être commandé par plusieurs systèmes.

Les ECC sont la généralisation des règles classiques *Évènement Condition Action* introduites dans le domaine des bases de données. Une ECC exprime un changement à entreprendre lorsqu'un évènement apparaît et que le système est dans un certain état. Plus exactement, lorsque l'évènement E apparaît et que la condition C est satisfaite, le changement C doit être entrepris.

Il existe deux types de changement :

- Le changement par action correspond à l'émission d'une action d'entrée d'un système, c'est-à-dire à l'émission par le contrôleur d'une commande à destination d'un dispositif. Ce type de règle se nomme *Évènement Condition Action* (ECA).
- Le changement par but correspond à un changement de l'état d'un système afin qu'il atteigne le but. Le but est un état du système. Dans ce cas, la série d'actions n'est pas énoncée afin d'atteindre l'état, c'est un calcul qui définit les actions à envoyer au dispositif. Ce type de règle se nomme *Évènement Condition But* (ECB).

Ce type de règle permet de décrire des comportements que les règles de contrainte ne peuvent pas exprimer. C'est notamment le cas pour les réactions aux évènements. En effet, les règles de contrainte permettent de gérer les états, mais pas les évènements. Par exemple, il est impossible de créer un comportement avec un bouton poussoir émettant un seul type d'évènement : « le bouton a été appuyé » en n'utilisant qu'un système de contraintes.

Une règle *Évènement Condition Action* est définie par un tuple : $eca = \langle \lambda^! \ , \ c \ , \ \alpha^? \rangle$, où $\lambda^! \in \Sigma^!$ est une action de sortie du système $\mathcal{S}1$ ($\Sigma^! \in \mathcal{S}1$), c est une condition booléenne sur les variables du système $\mathcal{S}2$ et $\alpha^? \in \Sigma^?$ est une action d'entrée du système $\mathcal{S}2$ ($\Sigma^? \in \mathcal{S}2$).

Une règle *Évènement Condition But* est définie par un tuple : $ecb = \langle \lambda^!, \ c, \ \gamma \rangle$, où $\lambda^! \in \Sigma^!$ appartient à l'automate de comportement du système $\mathcal{S}1$, c est une condition booléenne sur les variables du système $\mathcal{S}2$, et $\gamma \in S$ est un état du système $\mathcal{S}2$. Un état est défini par une localité $l \in L$ et les valeurs de chacune des variables de V.

5.3.2 Règles de contraintes

Les contraintes sont utilisées pour restreindre le comportement d'un système. L'expression d'une contrainte est écrite au moyen d'un langage logique. Nous limitons l'étude aux contraintes de comparaison entre les variables v_i d'un système et

des constantes. L'expression des contraintes est écrite en utilisant les opérateurs de comparaison ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$) et les opérateurs de l'algèbre booléenne ($\wedge$, $\vee$, $\neg$). Une contrainte c_1 s'écrit sous forme d'une série de disjonctions et conjonctions de comparaisons comme par exemple :

$$c_1 = v_1 < \text{cst}_1 \wedge v_2 \geq \text{cst}_2 \wedge \ldots$$

Rappelons qu'un système peut-être commandé par différents points de commande qui ne sont pas contrôlables. C'est le cas d'une télécommande d'une télévision qui ne peut pas être bridée puisqu'un utilisateur a toujours la possibilité de l'utiliser. C'est pourquoi un système peut violer une contrainte imposée. Dans ce cas, le contrôleur doit corriger l'état du système afin de garantir la contrainte. L'automate IOSTS du contrôleur est synthétisé à partir de l'automate de comportement du dispositif et de la règle de contrainte.

De la même manière que l'utilisateur peut, par un système de commande externe, violer une contrainte qui a été définie, le système peut démarrer dans un état qui viole cette contrainte. Dans ce cas, il n'y a pas d'état stable antérieur à l'état courant dans lequel le contrôleur pourrait ramener le système. Ansi, pour chaque contrainte, un état de secours est défini. Il correspond à l'état dans lequel le contrôleur doit amener le système si ce dernier démarre dans un état violant la contrainte.

5.4 IOSTS appliqué aux contrôleurs

L'état de l'art donne la définition d'un contrôleur comme un système $\mathcal{C}$ qui gère un autre système $\mathcal{S}$. Pour cela, il observe les évènements que $\mathcal{S}$ génère et autorise ou interdit l'utilisation d'actions d'entrée du système contrôlé en fonction des observations. En d'autres termes, le contrôleur limite l'expressivité des traces d'exécution à un sous-ensemble. Soit $\mathcal{S}_c$ le système $\mathcal{S}$ contrôlé, alots $Trace(\mathcal{S}_c) \subseteq Trace(\mathcal{S})$.

Dans le contexte des installations domotiques, un système de contrôle peut observer les évènements et interagir avec les objets communicants. En revanche, il lui est impossible d'interdire le déclenchement d'actions d'entrée. De plus, il est inenvisageable d'ajouter physiquement un élément interdisant les actions d'entrée des objets communicants. Dans la suite de ce mémoire, nous désignons par contrôleur un système gérant l'état d'un autre système sans possibilité de limiter l'expressivité des traces d'exécution du système contrôlé.

Un contrôleur est modélisé par un automate IOSTS particulier. En plus de disposer des localités virtuelles et des chemins d'entrée-sortie des automates de

comportement, les automates des contrôleurs disposent de localités transitoires. Un contrôleur est un système extérieur à l'installation, il émet des actions que le dispositif reçoit et reçoit des notifications que le dispositif émet. En d'autres termes, l'alphabet d'entrée (respectivement de sortie) du dispositif correspond à l'alphabet de sortie (d'entrée) du contrôleur.

Une localité transitoire l^t de l'ensemble des localités transitoires L^t ($l^t \in L^t$) est une localité correspondant à un ensemble d'états réels ou virtuels que l'utilisateur souhaite faire changer. Ces localités jalonnent les chemins d'entrée-sortie permettant au contrôleur de piloter les systèmes contrôlés.

Soit $\mathcal{C}$ un automate contrôleur et $L = \underline{L} \cup \overline{L^v}$ l'ensemble des localités stables et virtuelles. L'ensemble des localités transitoire L^t est composé de toutes les localités virtuelles et d'une partie des localités stables.

$$\overline{L^v} \subset L^t \subset L.$$

Un automate de contrôleur donne le comportement que doit suivre un contrôleur afin de répondre aux règles définies. Chaque transition de sortie qu'un contrôleur peut prendre doit être déclenchée afin de suivre les règles. En d'autres termes, lorsqu'un contrôleur est dans une localité avec la possibilité d'envoyer une commande, il doit l'envoyer. Le contrôleur permet de commander un dispositif. Pour cela, il utilise son automate de comportement afin de connaître l'état du dispositif. Un contrôleur est créé à partir de l'automate d'un système et de règles ECC ou de contraintes, l'opération de création se nomme synthèse de contrôleur. Nous verrons comment est synthétisé un contrôleur dans le chapitre suivant.

5.5 Synthèse

Les solutions domotiques présentées dans le chapitre 1 nous permettent de constater un manque dans le domaine de la gestion du comportement. Nous avons entrepris d'étudier dans le chapitre 3 les différents automates permettant de modéliser le comportement de systèmes, notamment les systèmes discrets à entrées/sorties. Dans ce chapitre, nous avons abordé les particularités des dispositifs domotiques et comment modéliser le comportement d'une installation domotique à partir de règles.

Les systèmes domotiques sont modélisés au moyen d'automates d'entrées/-sorties. Les commandes permettant d'utiliser les fonctionnalités des dispositifs correspondent aux actions d'entrée des automates et les notifications correspondent aux actions de sortie des automates. Les actions internes quant à elles formalisent la présence de multiples points de commande.

Les automates IOSTS des dispositifs bénéficient de propriétés particulières. En effet, nous avons introduit les localités virtuelles qui modélisent la non-connaissance de l'état d'un système. Cet état n'est pas réel pour le dispositif, mais caractérise l'attente d'une réponse pour un système extérieur. Dans la plupart des cas, une commande envoyée à un système domotique est suivie d'une notification de retour permettant de connaître le nouvel état du système. Nous l'avons appelé chemin d'entrée-sortie. Ces chemins sont composés d'une transition d'entrée menant à une localité virtuelle et d'une transition de sortie ayant comme source cette localité virtuelle.

Le comportement d'une installation domotique est la composition de tous les comportements des dispositifs occupant l'installation. Afin de domestiquer ce comportement, nous avons introduit deux types d'expressions, les règles de changements sur conditions (ECC) et les règles de contraintes. Les règles ECC ont un aspect dynamique et créent un changement dans une installation lorsqu'un évènement intervient et que l'installation est dans un état particulier. Les règles de contrainte ont un aspect statique et créent un changement seulement lorsque l'installation entre dans un état particulier.

Sommaire

Chapitre

6

Contrôle des objets communicants

Dans la théorie du contrôle, les contrôleurs, ou superviseurs, sont présents afin de limiter le comportement possible d'un système. Pour cela, ils interdisent le déclenchement d'actions. Les superviseurs des automates à entrées/sorties sont capables d'interdire le déclenchement d'actions d'entrée d'un système. Dans le cas d'une installation domotique, cela correspondrait au fait d'interdire à un dispositif de la maison de recevoir une commande. De plus, il n'est pas possible de limiter le nombre de points de commande d'une installation ni de les exclure ou de les reprogrammer. De ce fait, il est impossible d'interdire l'émission de commande à destination des dispositifs. Il n'est pas envisageable d'utiliser un superviseur tel qu'il est défini dans la théorie du contrôle.

En revanche, nous utilisons le terme contrôleur pour désigner un système permettant d'empêcher que le système demeure dans certains états. Pour cela, le contrôleur est capable d'utiliser le comportement des dispositifs d'une installation domotique et il est capable d'observer tous les changements d'état des systèmes domotiques. C'est-à-dire que le contrôleur peut émettre des commandes et recevoir des notifications des dispositifs.

La synthèse d'un contrôleur nécessite d'énoncer des règles. Nous avons introduit dans le chapitre précédent des règles permettant de déclencher un comportement :

- lors de l'apparition d'un évènement (ECC) ;
- lorsqu'un dispositif atteint un état non souhaitable (contrainte).

6.1 Expression du besoin

Une installation domotique correspond à un ensemble de dispositifs commandables à distance tels que les lumières, les volets, les télévisions, etc.. Une installation domotique est, elle aussi, un système ; son comportement correspond à la somme de tous les comportements des dispositifs qui la composent. Les dispositifs d'une installation interagissent ensemble de diverses manières et ne sont pas dépendants les uns des autres. Le comportement de chaque dispositif est modélisé comme un ensemble de transitions entre états.

Soit $\mathcal{S}$ l'automate IOSTS du système d'une installation domotique avec un nombre fini de localités. L'automate $\mathcal{S}$ modélise le « comportement non contrôlé » de l'installation. Le comportement non contrôlé signifie que les changements d'état interviennent seulement lors de la réception de commandes émises depuis un point de commande. Ce comportement a de grandes chances de ne pas être satisfaisant pour un utilisateur et doit être adapté. La modification du comportement de dispositifs n'est pas possible. En revanche, il est possible de mettre en place un système de « pilotage automatique ».

Dans le but de piloter le comportement d'un système $\mathcal{S}$, nous introduisons un automate contrôleur $\mathcal{C}$. Ce qui soulève deux questions :

- Comment sont créés les contrôleurs ?
- Comment un contrôleur $\mathcal{C}$ agit sur le comportement de $\mathcal{S}$?

Le but est de produire un système de contrôle d'une installation domotique basé sur la rigueur d'un formalisme à base d'automate à entrées-sorties. Nous avons expliqué dans le chapitre précédent le formalisme des systèmes domotique et des systèmes de contrôle. La création du système de contrôle est réalisée par un outil mathématique prenant en compte une installation domotique et des règles de comportement.

Certain états de $\mathcal{S}$ ne sont pas désirables et doivent être évités. Par exemple, un chauffage allumé dans une pièce où une fenêtre est ouverte ou un volume trop élevé d'une télévision lorsque le téléphone sonne. Afin de caractériser ces états, nous introduisons le terme d'états illégaux. Inversement, les états désirables sont nommés états légaux.

Dans notre modèle, $\mathcal{C}$ observe tous les évènements émis par $\mathcal{S}$. Lorsque $\mathcal{S}$ entre dans un état illégal, $\mathcal{C}$ agit sur $\mathcal{S}$ afin de le ramener à un état légal. La décision de l'état légal à atteindre est réalisée par un algorithme de choix. Par exemple, cet algorithme peut prendre en compte l'état légal précédent, un état de secours, etc..

D'un point de vue de la théorie du contrôle, les installations domotiques sont complexes à cause :

– du caractère réparti de l'architecture des dispositifs ;
– du nombre de dispositifs et de l'hétérogénéité des types de dispositifs ;
– de la complexité du comportement de plusieurs dispositifs.

Nous faisons le postulat que tant que l'on ne soumet pas l'installation domotique à des règles de comportement, ce système ne possède pas d'interblocage ni aucune boucle infinie. C'est-à-dire que le système est commandable à tout instant.

6.2 Opération sur les automates

Les opérateurs présentés dans cette section sont décrits au moyen de spécifications basées sur le langage naturel. La plupart du temps, ils sont accompagnés de leur représentation graphique.

6.2.1 Localité illégale ou partiellement illégale

Une localité est illégale (on note $Illegale(l)$)lorsque les états qu'elle représente sont illégaux. Un état est illégal lorsqu'une règle de contrainte l'affirme. Celui-ci n'est pas souhaitable pour une installation, c'est pourquoi le contrôleur doit mettre en œuvre des actions afin de sortir l'installation de cet état.

De façon similaire, une localité l est légale (noté $Legale(l)$) si tous ses états sont légaux.

Une localité l partiellement illégale (noté $PIllegale(l)$) est une localité pour laquelle une partie des états $S_i \subset l$ sont illégaux et l'autre partie $S_l \subset l$ sont légaux avec $S_i \cup S_l = l \wedge S_i \cap S_l = \emptyset$. Il est possible d'utiliser la division de localités (cf. section 6.2.4) sur l afin d'obtenir deux localités l_1 et l_2 telles que :

$$Div(l, c) = \langle l_1, l_2, T \rangle$$

avec la contrainte c définie de tel manière que $S_i = l_1$ et $S_l = l_2$.

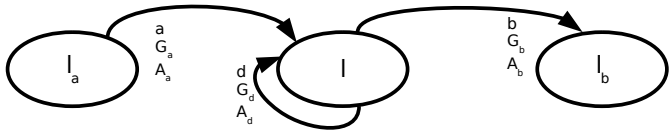
l_1 ainsi produite est une localité illégale et l_2 est une localité légale.

6.2.2 Miroir

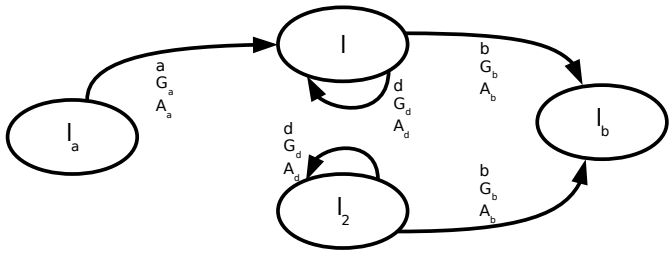
Soit les systèmes :

$$\mathcal{S}_1 = \langle D_{\mathcal{S}_1}, \Theta_{\mathcal{S}_1}, L_{\mathcal{S}_1}, L^0_{\mathcal{S}_1}, \Sigma_{\mathcal{S}_1}, T_{\mathcal{S}_1} \rangle \text{ et } \mathcal{S}_2 = \langle D_{\mathcal{S}_2}, \Theta_{\mathcal{S}_2}, L_{\mathcal{S}_2}, L^0_{\mathcal{S}_2}, \Sigma_{\mathcal{S}_2}, T_{\mathcal{S}_2} \rangle$$

L'opération $Miroir$ permet d'inverser les actions d'entrée et les actions de sortie. Ainsi $Miroir(\mathcal{S}_1) = \mathcal{S}_2$ donne le système $\mathcal{S}_2$ tel que :



(a) Automate d'une localité à dupliquer



(b) Automate résultat de la duplication d'une localité

FIGURE 6.1 – Duplication d'une localité $Dup(l) = \langle l_2, T \rangle$

$$
\begin{array}{lcl}
D_{\mathcal{S}_1} & = & D_{\mathcal{S}_2} \\
\Theta_{\mathcal{S}_1} & = & \Theta_{\mathcal{S}_2} \\
L_{\mathcal{S}_1} & = & L_{\mathcal{S}_2} \\
L^0_{\mathcal{S}_1} & = & L^0_{\mathcal{S}_2}
\end{array}
\qquad
\begin{array}{lcl}
\Sigma^?_{\mathcal{S}_1} & = & \Sigma^!_{\mathcal{S}_2} \\
\Sigma^!_{\mathcal{S}_1} & = & \Sigma^?_{\mathcal{S}_2} \\
\Sigma^\tau_{\mathcal{S}_1} & = & \Sigma^\tau_{\mathcal{S}_2} \\
T_{\mathcal{S}_1} & = & T_{\mathcal{S}_2}
\end{array}
$$

6.2.3 Duplication de localités

La duplication d'une localité l d'un système $\mathcal{S}$ produit une nouvelle localité l_2. Elle représente le même ensemble d'états que l. Une duplication s'exprime par l'opérateur Dup :

$$Dup(l) = \langle l_2, T \rangle$$

où :

- l correspond à la localité à dupliquer ;
- l_2 est la localité produite ;
- T est l'ensemble des transitions produites par l'opération.

Tous les cas possibles lors de la duplication d'une localité l sont présentés dans la figure 6.1. La figure 6.1(a) est l'automate comportant la localité l ainsi que les transitions et les localités voisines ; La figure 6.2(b) correspond au résultat de la duplication.

Les transitions t' de la nouvelle localité sont créées afin de correspondre aux transitions t ayant comme source l. C'est le cas des transitions étiquetées b et d

dans la figure 6.1.

$$\forall t = \langle l, a, G, A, l_a \rangle \in T,\ \exists\, t' \in T \mid t' = \langle l_2, a, G, A, l_a \rangle$$

Les transitions ayant comme destination l et n'ayant pas comme source l ne sont pas reportées pour la localité l_2. C'est le cas des transitions étiquetées a dans la figure 6.1.

$$\forall t = \langle l_a, a, G, A, l \rangle \in T \mid l \neq l_a,\ \nexists\, t' \in T \mid t' = \langle l_a, a, G, A, l_2 \rangle$$

La localité l_2 ne possède pas de transition entrante (hormis celles réflexives), c'est à dire des transitions pour lesquelles l_2 est destination ; C'est à l'utilisateur de l'opérateur *Dup* de spécifier les transitions à destination de la localité dupliquée.

6.2.4 Division de localités

La division d'une localité l d'un système $\mathcal{S}$ produit deux localités l_1 et l_2 chacune représentant un sous-ensemble des états de l. Une division s'exprime par l'opérateur *Div* :

$$Div(l, c) = \langle l_1, l_2, T \rangle$$

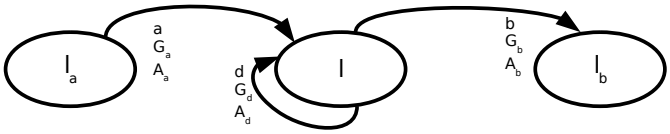
avec :

- l la localité à diviser ;
- c la contrainte de division ;
- l_1 la localité correspondant aux états satisfaisant la condition c ;
- l_2 la localité correspondant au reste des états de l ;
- T l'ensemble des transitions produites par l'opération.

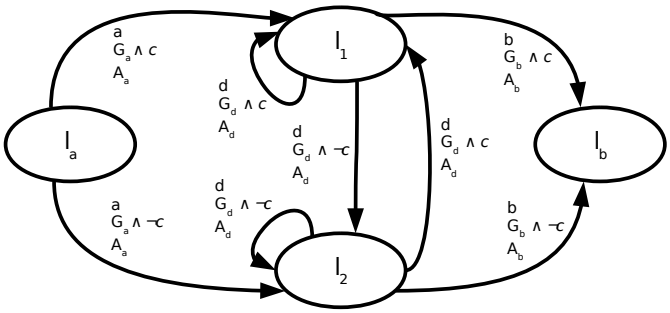
Les états de l_1 sont départagés des états de l_2 au moyen d'une règle c exprimée sur les valeurs des variables du système sous forme de logique booléenne. Les transitions $t \in T$ des deux nouvelles localités sont créees afin de correspondre aux transitions de l, leur garde est ajustée en fonction de la règle de division.

Tous les cas possibles lors de la division d'une localité sont présentés dans la figure 6.2. La figure 6.2(a) est un automate comportant la localité l ainsi que des transitions et des localités voisines ; 6.2(b) correspond au résultat de la division.

Lorsqu'une transition $t = \langle l, a, G, A, l \rangle$ possède comme source et destination une localité à diviser l, quatre transitions t_1, t_2, t_3 et t_4 sont créées avec la même action, les mêmes affectations et une garde ajustée :



(a) Automate d'une localité à diviser



(b) Automate résultat de la division d'une localité

FIGURE 6.2 – Division d'une localité $Div(l, c) = \langle l_1, l_2, T \rangle$

- Deux transitions en direction de l_1 sont créées avec la condition c : $t_i = \langle l_i, a, G \wedge c, A, l_1 \rangle$ pour $i \in \{1, 2\}$.
- Deux transitions en direction de l_2 sont créées avec la négation de la condition c : $t_{(i+2)} = \langle l_i, a, G \wedge \neg c, A, l_2 \rangle$ pour $i \in \{1, 2\}$.

C'est le cas des transitions portant l'action d de la figure 6.2.

L'opération Div produit deux transitions t_1 et t_2 à partir des transitions $t = \langle l, a, G, A, l_a \rangle$ ayant pour source l telles que : $t_1 = \langle l_1, a, G \wedge c, A, l_b \rangle$ et $t_2 = \langle l_2, b, G \wedge \neg c, A, l_b \rangle$. C'est le cas des transitions b de la figure 6.2.

Enfin, l'opération Div produit deux transitions t_3 et t_4 à partir des transitions $t = \langle l_a, a, G, A, l \rangle$ ayant pour destination l telles que : $t_3 = \langle l_a, a, G \wedge c, A, l_1 \rangle$ et $t_4 = \langle l_a, a, G \wedge \neg c, A, l_2 \rangle$. C'est le cas des transitions a de la figure 6.2.

6.2.5 Somme de comportements

Les systèmes IOSTS représentent le comportement des dispositifs domotiques. Ces dispositifs sont des éléments d'une installation domotique. Le comportement d'une installation est composé des comportements des dispositifs qui la constituent. C'est la somme des automates des dispositifs, l'opération est notée $\oplus$.

Les opérations présentées dans la section 3.2.3 (page 56), ont pour but de faire correspondre un automate d'un système à un automate d'un contrôleur afin d'effectuer des tests ou de valider des propriétés. La somme de comportement

est une opération de composition de deux IOSTS. À l'inverse du produit et de l'opération parallèle, la somme de comportement crée un système regroupant deux systèmes indépendants. Cette somme est utilisée afin de synthétiser un contrôleur à partir de plusieurs. La somme de comportement permet d'effectuer des calculs sur un système virtuel qui est le résultat de la composition de différents systèmes indépendants les uns des autres.

Les actions des automates IOSTS sont des évènements. Les évènements sont traités les uns à la suite des autres ; il est donc impossible d'avoir deux actions déclenchées en même temps. Les transitions des automates IOSTS conviennent afin de modéliser le comportement global d'une installation. En effet, ces transitions disposent seulement d'une seule action qui permet de la déclencher.

Dans la suite de cette sous-section, nous considérerons trois systèmes où tous les états sont accessibles :

$$\begin{aligned} \mathcal{S}_1 &= \langle D_{\mathcal{S}_1}, \Theta_{\mathcal{S}_1}, L_{\mathcal{S}_1}, L^0_{\mathcal{S}_1}, \Sigma_{\mathcal{S}_1}, T_{\mathcal{S}_1} \rangle, \\ \mathcal{S}_2 &= \langle D_{\mathcal{S}_2}, \Theta_{\mathcal{S}_2}, L_{\mathcal{S}_2}, L^0_{\mathcal{S}_2}, \Sigma_{\mathcal{S}_2}, T_{\mathcal{S}_2} \rangle, \\ \mathcal{S} &= \langle D_{\mathcal{S}}, \Theta_{\mathcal{S}}, L_{\mathcal{S}}, L^0_{\mathcal{S}}, \Sigma_{\mathcal{S}}, T_{\mathcal{S}} \rangle. \end{aligned}$$

Nous imposons que $\mathcal{S}_1$ et $\mathcal{S}_2$ soient indépendants, c'est-à-dire qu'aucune variable, aucune action et aucune localité ne soit commune aux deux systèmes :

$$D_{\mathcal{S}_1} \cap D_{\mathcal{S}_1} = \emptyset, L_{\mathcal{S}_1} \cap L_{\mathcal{S}_2} = \emptyset \text{ et } \Sigma_{\mathcal{S}_1} \cap \Sigma_{\mathcal{S}_2} = \emptyset.$$

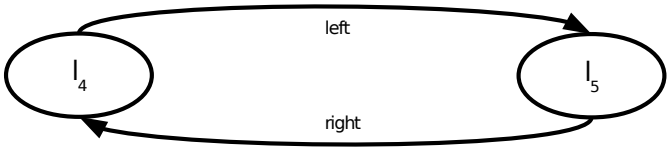
La somme des deux systèmes $\mathcal{S}_1$ et $\mathcal{S}_2$ est l'automate $\mathcal{S} = \mathcal{S}_1 \oplus \mathcal{S}_2$. Il est l'union des variables des deux systèmes, l'union des conditions initiales et l'union des actions. De plus, la somme de deux systèmes est le produit cartésien des localités et des localités initiales. Enfin, la somme projette les transitions de chacun des systèmes dans l'espace de ses localités $L_{\mathcal{S}}$.

Plus formellement $\mathcal{S} = \mathcal{S}_1 \oplus \mathcal{S}_2$ est définie par :

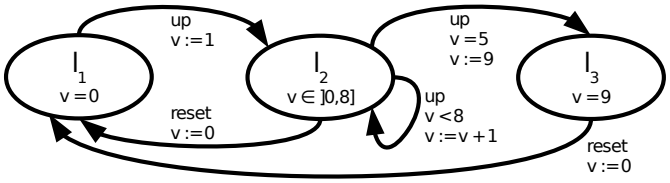
$$\begin{aligned} D_{\mathcal{S}} &= D_{\mathcal{S}_1} \cup D_{\mathcal{S}_2}, \\ \Theta_{\mathcal{S}} &= \Theta_{\mathcal{S}_1} \wedge \Theta_{\mathcal{S}_2}, \\ L_{\mathcal{S}} &= L_{\mathcal{S}_1} \times L_{\mathcal{S}_2}, \\ L^0_{\mathcal{S}} &= L^0_{\mathcal{S}_1} \times L^0_{\mathcal{S}_2}, \\ \Sigma_{\mathcal{S}} &= \Sigma_{\mathcal{S}_1} \cup \Sigma_{\mathcal{S}_2}, \\ T_{\mathcal{S}} &= f(T_{\mathcal{S}_1}, L_{\mathcal{S}_2}) \cup f(T_{\mathcal{S}_2}, L_{\mathcal{S}_1}), \end{aligned}$$

où f est la fonction de modification des transitions qui crée pour chacune des transitions de l'ensemble passé en paramètre, les transitions entre les localités de l'ensemble $L_{\mathcal{S}}$ de manière à projeter les transitions des automates $\mathcal{S}_1$ et $\mathcal{S}_2$ dans $\mathcal{S}$. $f(T, L) = X$ est un ensemble de transitions tel quel :

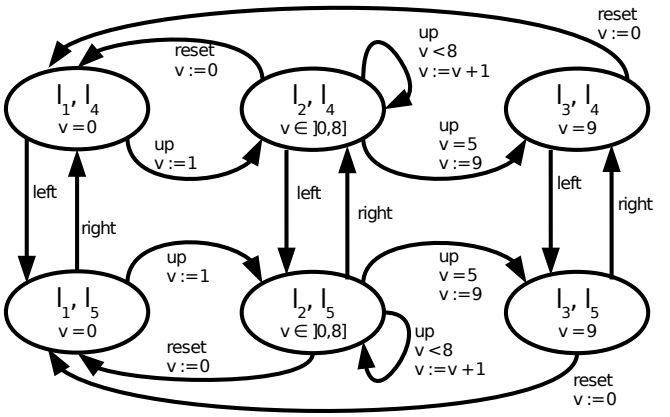
$$\forall t = \langle l^o, a, G, A, l^d \rangle \in T,\ \forall l \in L,\ x = \left\langle \langle l^o, l \rangle, a, G, A, \langle l^d, l \rangle \right\rangle \in X.$$



(a) Automate du système $\mathcal{S}_1$



(b) Automate du système $\mathcal{S}_2$



(c) Automate de la somme des deux systèmes $\mathcal{S}_1 \oplus \mathcal{S}_2$

FIGURE 6.3 – Somme des deux systèmes $\mathcal{S}_1 \oplus \mathcal{S}_2$

La figure 6.3 donne un exemple de la somme de deux systèmes. Ces deux systèmes sont présentés par la figure 6.3(a) et 6.3(b). Comme la somme des automates ne change pas le type d'actions, les automates ne sont pas représentés avec le type d'actions (sortie, entrée ou interne), il n'y a donc pas de symbole « $?, !, \tau$ » étiqueté sur les flèches. Le système $\mathcal{S}_1$ est un dispositif à deux états avec deux actions. Le système $\mathcal{S}_2$ est plus complexe et il dispose de trois localités et d'une variable v. Les figures 6.3(b) et 6.3(c) montrent le domaine de la variable juste en dessous du nom de la localité.

Le résultat est le produit cartésien des localités des deux automates. Les transitions présentent dans le résultat $\mathcal{S}$ permettent aux systèmes $\mathcal{S}_1$ et $\mathcal{S}_2$ d'évoluer indépendamment l'un de l'autre. Rappelons que les actions sont des évènements, et ainsi elles ne peuvent se produire en même temps. Pour chaque transition d'un système $\mathcal{S}_i$ avec $i = 1, 2$, on retrouve une ou plusieurs transitions avec la même action, la même garde et la même affectation dans le système $\mathcal{S}$. En revanche, les localités sources et destinations sont changées, car elles permettent de prendre en compte le deuxième système.

6.3 Synthèse du contrôleur d'objets communicants

La création d'un contrôleur d'objets communicants est réalisée à partir de règles de changement sur évènements ou des règles de contraintes. L'algorithme de synthèse change en fonction du type de règles.

Lorsque la règle de contrainte concerne plusieurs dispositifs, l'opération de somme de comportement doit être réalisée entre les automates des dispositifs. Cette opération permet d'avoir un seul et unique système. La règle est alors appliquée au système créé.

6.3.1 Comportement sur évènement

Nous présentons ici les algorithmes permettant de synthétiser un contrôleur de système avec des règles de changement ECC :

- Les règles de changement par actions : Évènement Condition Action (ECA).
- Les règles de changement par but : Évènement Condition But (ECB).

Comportement d'action

L'algorithme synthétisant le contrôleur produit un automate IOSTS $\mathcal{C}$. Ce contrôleur est synthétisé à partir d'une règle ECA $\mathcal{R} = \langle \lambda^!,\ c,\ \alpha^? \rangle$ et de deux automates de comportement $\mathcal{S}1 = \langle D_{\mathcal{S}_1}, \Theta_{\mathcal{S}_1}, L_{\mathcal{S}_1}, \underline{L_{\mathcal{S}_1}}, \Sigma_{\mathcal{S}_1}, T_{\mathcal{S}_1} \rangle$ et $\mathcal{S}2 = \langle D_{\mathcal{S}_2}, \Theta_{\mathcal{S}_2},$

$L_{\mathcal{S}_2}, \underline{L_{\mathcal{S}_2}}, \Sigma_{\mathcal{S}_2}, T_{\mathcal{S}_2}\rangle$. $\lambda^!$ est une actions de sortie de $\mathcal{S}2$ pour laquelle les paramètres, s'il y en a, ont été affectés. $\alpha^?$ est une actions d'entrée de $\mathcal{S}1$ pour laquelle les paramètres, s'il y en a, ont été affectés.

L'algorithme de synthèse crée un automate $\mathcal{C} = \langle D_{\mathcal{C}}, \Theta_{\mathcal{C}}, L_{\mathcal{C}}, \underline{L_{\mathcal{C}}}, \Sigma_{\mathcal{S}1} \cup \{\lambda^!\}, T_{\mathcal{C}}\rangle$ tel que :

- Les localités stables de $\mathcal{S}1$ appartiennent à $\mathcal{C}$.

$$\underline{L_{\mathcal{C}}} = \underline{L_{\mathcal{S}_1}}$$

- Les transitions de sortie de $\mathcal{S}1$ ayant comme origine les localités stables appartiennent à $\mathcal{C}$; Les transitions de sortie sont liées aux mêmes localités que dans le système $\mathcal{S}1$.

$$\forall t = \langle l_1, a, G, A, l_2\rangle \in T_{\mathcal{S}_1} \mid a \in \Sigma^!_{\mathcal{S}_1} \ \wedge \ l_1 \in \underline{L_{\mathcal{S}_1}} \Longrightarrow t \in T_{\mathcal{C}}.$$

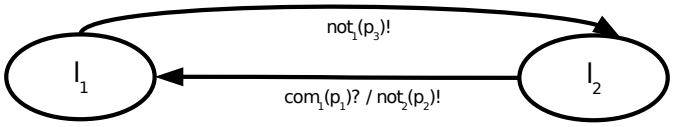
- Les transitions d'entrée t_e associées à l'action $\alpha^?$ de $\mathcal{S}1$ sont copiées et forment un ensemble $T' \in \mathcal{C}$. Les paramètres $sig(\alpha^?)$ des transitions de T' sont donnés par la condition de la règle $\mathcal{R}$. Chacune des sources l_1 des transitions d'entrée t_e est dupliquée. Ces *duplicatas* l_a sont des localités transitoires et stables ($l_a \in \underline{L} \cap L^t$). Chaque duplicata l_a est la source des transitions d'entrée copiées $t_2 \in T'$. De plus, ces localités l_a sont la destination de transitions t_2 déclenchées par l'action $\lambda^!$. La destination de la transition t_2 est la destination de la transition d'entrée t_e.

  La garde c de la transition copiée est la conjonction de la garde de la règle $\mathcal{R}$ et d'une partie de la garde de la transition de notification qui la suit. L'adaptation de la garde est nécessaire pour obtenir les transitions $t \in T'$

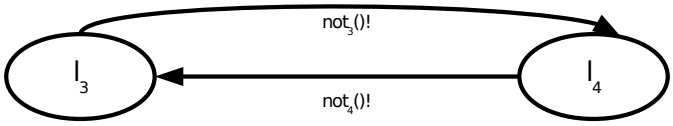
$$\begin{array}{l}
\forall t_e = \langle l_1, a_e, G_e, A_e, l_2\rangle \in T_{\mathcal{S}_1} \mid \alpha^? \in a_e \wedge a_e \in \Sigma^?_{\mathcal{S}_1}, \\
\exists t_s = \langle l_2, a_s, G_s, A_s, l_3\rangle \in T_{\mathcal{S}_1} \mid a_s \in \Sigma^!_{\mathcal{S}_1}, \\
\exists l_a, (l_a, T') = Dup(l_2), \\
\quad l_a \in L^t_{\mathcal{C}}, l_2 \in \overline{L^v_{\mathcal{C}}} \\
\quad \lambda^! \in \Sigma^!_C, a_e \in \Sigma^?_C, a_s \in \Sigma^!_C, \\
\quad t_1 = \langle l_1, \lambda^!, g(G_e, \mathcal{R}), \emptyset, l_a\rangle \in T_{\mathcal{C}}, \\
\quad t_2 = \langle l_a, a_e, G', A_e, l_2\rangle \in T_{\mathcal{C}} \text{ avec } G' = condition_{\alpha}(\mathcal{R}), \\
\quad t_3 = \langle l_2, a_s, G_s, A_s, l_3\rangle \in T_{\mathcal{C}}, \\
\quad t_4 = \langle l_2, \tau, true, \emptyset, l_a\rangle \in T_{\mathcal{C}}.
\end{array}$$

La fonction $g(G, \mathcal{R})$ est une garde qui retourne vrai si les gardes de G_e et c sont valides selon le contexte et les valeurs des paramètres de $\alpha^?$.

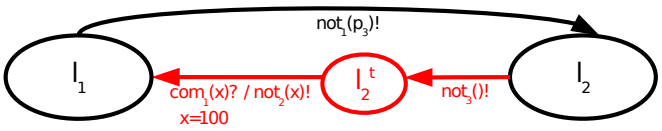
$$g(G_e, \mathcal{R}) == G \wedge c \wedge condition_{\alpha}(\mathcal{R}) \wedge condition_{\lambda}(\mathcal{R})$$



(a) Automate de comportement du système commandé $\mathcal{S}1$



(b) Automate de comportement du point de commande $\mathcal{S}2$



(c) *Miroir* de l'automate du contrôleur de la règle ECA $\mathcal{C}$

FIGURE 6.4 – Synthése d'un contrôleur d'un dispositif fictif avec une règle ECA $\langle not_3()!,\ c,\ com_1(100)?\rangle$.

La fonction $condition_{\#}$ $(\mathcal{R} \rightarrow G)$ avec $\# \in \{\lambda, \alpha\}$ associe pour la règle ECA, la garde sur les paramètres de l'action $\#$ conditionné par les valeurs données dans la règle ECA. Par exemple, le résultat de $conditionAlpha_{\alpha}(\langle vnot(), val = 3, cmd(40, 50)\rangle)$ donnera la condition $p_1 = 40 \wedge p_2 = 50$ lorsque p_1 et p_2 sont les symboles des paramètres de l'action cmd.

La fonction $adapterCondition$ $(G \rightarrow G)$ filtre, pour un prédicat donné, les conditions portant uniquement sur les variables. Une condition G peut-être vue comme un ensemble de conjonctions de clauses. Cette fonction garde les clauses impliquant uniquement les variables du système et pas les paramètres des actions. Par exemple, un prédicat $adapterCondition(\text{« } val > 60 \ \wedge \ p < 100 \ \wedge \ val > p \text{ »})$ donne le résultat « $val > 60$ ».

L'opération *Miroir* est utilisée à la fin de l'algorithme afin d'inverser les actions d'entrée et les actions de sortie. En effet, le contrôleur envoie les commandes au système (actions de sortie pour le contrôleur) alors que le système les reçoit (actions d'entrée pour le système) et inversement, le contrôleur reçoit les notifications du système (actions d'entrée) alors que le système les envoie (actions de sortie).

Par exemple, la figure 6.4 illustre la synthèse d'un contrôleur issu d'une règle ECA, d'un dispositif et du point de commande qui lui est associé. La figure 6.4(a) présente le système à contrôler, la figure 6.4(b) décrit le point de commande et la figure 6.4(c) décrit le *Miroir* du contrôleur synthétisé.

Les différences entre le contrôleur $\mathcal{C}$ et le dispositif $\mathcal{S}_1$ sont :

- l'ajout d'une localité transitoire l_2^t ;
- l'ajout d'une transition d'entrée portant sur l'alphabet du point de commande (6.4(b)) et ayant comme garde c.
- l'ajout d'une transition d'entrée-sortie permettant de sortir de la localité transitoire ;
- la suppression des transitions d'entrée-sortie de $\mathcal{S}1$.

Comportement de but

L'algorithme synthétisant le contrôleur produit un automate IOSTS contrôleur $\mathcal{C}$. Ce contrôleur est synthétisé à partir d'une règle ECB $\langle \lambda^!,\ c,\ \gamma \rangle$ et des automates de comportement $\mathcal{S}1$ et $\mathcal{S}2$ tel que $\lambda^!$ appartient aux actions de sortie de $\mathcal{S}2$ et γ est couple représentant un état de $\mathcal{S}1$.

Les automates sont définis par :

$$\begin{aligned}\mathcal{S}1 &= \langle D_{\mathcal{S}_1}, \Theta_{\mathcal{S}_1}, L_{\mathcal{S}_1}, \underline{L_{\mathcal{S}_1}}, \Sigma_{\mathcal{S}_1}, T_{\mathcal{S}_1} \rangle \\ \mathcal{S}2 &= \langle D_{\mathcal{S}_2}, \Theta_{\mathcal{S}_2}, L_{\mathcal{S}_2}, \underline{L_{\mathcal{S}_2}}, \Sigma_{\mathcal{S}_2}, T_{\mathcal{S}_2} \rangle \\ \mathcal{C} &= \langle D_{\mathcal{C}}, \Theta_{\mathcal{C}}, L_{\mathcal{C}}, \underline{L_{\mathcal{C}}}, \Sigma_{\mathcal{S}_1} \cup \{\lambda^!\}, T_{\mathcal{C}} \rangle\end{aligned}$$

L'algorithme de synthèse crée le miroir de l'automate contrôleur $\mathcal{C}$ tel que :

- Les localités stables de $\mathcal{S}1$ sont celles de $\mathcal{C}$.

$$\underline{L_{\mathcal{C}}} = \underline{L_{\mathcal{S}_1}}$$

- Les transitions de sortie de $\mathcal{S}1$ ayant comme origine les localités stables appartiennent à $\mathcal{C}$; Les transitions de sortie sont liées aux mêmes localités que dans le système $\mathcal{S}1$.

$$\forall t = \langle l_1, a, G, A, l_2 \rangle \in T_{\mathcal{S}_1} \mid a \in \Sigma^!_{\mathcal{S}_1} \ \wedge\ l_1 \in \underline{L_{\mathcal{S}_1}} \Longrightarrow t \in T_{\mathcal{C}}.$$

- Pour chaque localité stable un chemin est calculé vers le but γ. Les transitions des chemins calculés sont copiées dans $\mathcal{C}$ ainsi que les localités intermédiaires. Les sources des premières transitions du chemin sont dupliquées afin d'être des localités transitoires du système $\mathcal{C}$. Les localités transitoires dupliquées sont la destination de transitions déclenchées par l'action $\lambda^!$ du système $\mathcal{S}2$ et ayant comme garde c. La source de cette dernière transition est la source de la première transition du chemin calculé.

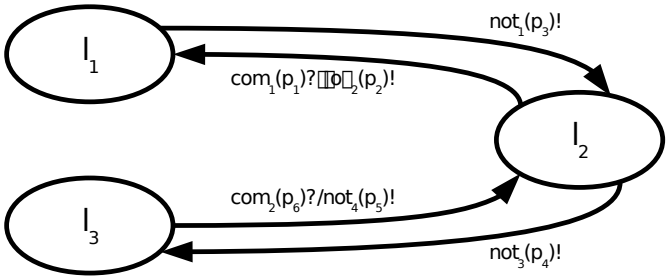
$$
\begin{array}{l}
\forall l \in \underline{L_{\mathcal{C}}} \\
\quad T' = trouverCheminJusquAEtat(l, \gamma) \\
\quad (L'', T'') = dupliquerCheminVersEtat\mathcal{C}(l, T') \\
\quad L'' \subset L^t_{\mathcal{C}} \wedge T'' \subset T_{\mathcal{C}} \\
\quad \exists (l^s, T^s) = Dup(l) \text{ avec } l^s \in l^t_{\mathcal{C}} \\
\quad \exists t_p = \langle l, a_p, G_p, A_p, l' \rangle \mid [t_p | T_q] = T'' \\
\quad \exists t_s = \langle l, \lambda^!, g(G_p, \mathcal{R}), \emptyset, l^s \rangle \in T_{\mathcal{C}}
\end{array}
$$

La fonction $trouverCheminJusquAEtat$ $(L_{\mathcal{S}} \times (L_{\mathcal{S}}, \Theta) \rightarrow 2^{T_{\mathcal{S}}})$ est définie par $trouverCheminJusquAEtat(l^s, (l^d, \Theta^d)) = T$. $2^{L_{\mathcal{C}}}$ désigne l'ensemble des parties de l'ensemble $L_{\mathcal{C}}$. La localité l^s représente la source du chemin et le couple (l^d, Θ^d) représente l'état destination. Cette fonction associe pour des paramètres d'entrée un ensemble de transitions non internes $T \in 2^{T_{\mathcal{S}}}$ permettant d'amener le dispositif à l'état destination. La recherche commence par les transitions d'entrée ayant comme origine l^s. L'algorithme consiste en une recherche de chemin dans un graphe, soit $\mathcal{S}1$ dans notre cas. Notre implantation utilise un parcours en largeur d'abord de $\mathcal{S}1$ en affectant toutes les combinaisons de valeurs possibles aux paramètres des transitions.

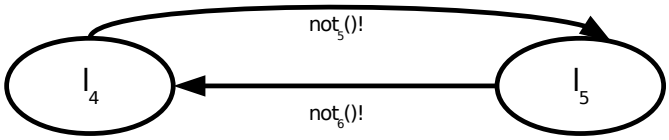
La fonction $dupliquerCheminVersEtat$ $(L_{\mathcal{S}} \times 2^{T_{\mathcal{S}}} \rightarrow 2^L \times 2^T)$ est définie par $f(l^s, T') = (L, T)$. La localité l^s représente la source du chemin créé par la fonction et (L, T) représente un couple d'un ensemble de localités et d'un ensemble de transitions. Cette fonction crée les transitions du chemin et les localités intermédiaires compatibles avec $\mathcal{C}$, le résultat devant être intégré dans le système $\mathcal{C}$.

$$
f(l^s, T) = \left\{
\begin{array}{ll}
t^d = \langle l^s, a, G, A, l^b \rangle & \text{Si } T = [\, t^h \mid [] \,] \\
(\emptyset, \{t^d\}) & \quad \text{avec } t^h = \langle l^a, a, G, A, l^b \rangle \\
 & \\
(l^d, T^d) = Dup(l^b) & \\
l^d \in W(l^s) & \\
(L', T') = f(l^d, T^q) & \text{Si } T = [t^h | T^q] \text{ avec } T^q \neq [] \\
t^d = \langle l^s, a, G, A, l^d \rangle & \quad \text{et } t^h = \langle l^a, a, G, A, l^b \rangle \\
\text{Si } a \in \Sigma^? & \\
\quad t^\tau = \langle l^d, \tau, true, \emptyset, l^s \rangle & \\
\quad (L' \cup \{l^d, l^\tau\}, T' \cup \{t^d\}) & \\
\text{Sinon} & \\
\quad (L' \cup \{l^d\}, T' \cup \{t^d\}) & \\
\text{Fin Si} &
\end{array}
\right.
$$

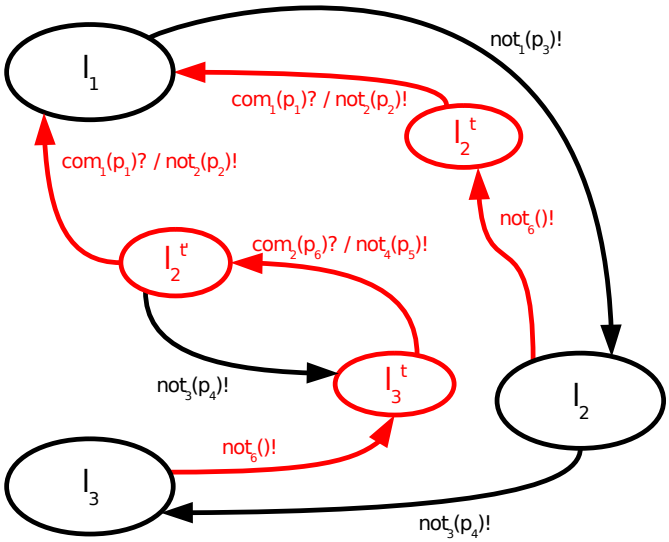
L'opération $Miroir$ est utilisée à la fin de l'algorithme afin d'inverser les actions d'entrée et les actions de sortie.



(a) Automate de comportement du système commandé $\mathcal{S}1$



(b) Automate de comportement du point de commande $\mathcal{S}2$



(c) *Miroir* de l'automate du contrôleur de la règle ECB $\mathcal{C}$

FIGURE 6.5 – Synthèse d'un contrôleur d'un dispositif fictif avec une règle ECB $\langle not6()!\ ,\ c\ ,\ \gamma\rangle$ avec $l_1 = \gamma$.

Par exemple, la figure 6.4 illustre la synthèse d'un contrôleur issu d'une règle ECB, d'un dispositif et du point de commande qui lui est associé. La figure 6.4(a) présente le système à contrôler, la figure 6.4(b) décrit le point de commande et la figure 6.4(c) décrit le *Miroir* du contrôleur synthétisé.

Les différences entre le contrôleur (6.4(c)) et le dispositif (6.4(a)) sont :

- l'ajout des localités transitoires $l_2^t, l_2^{t'}, l_3^t$;
- l'ajout de transitions d'entrée portant sur l'alphabet du point de commande (6.4(b)) et ayant comme garde c à partir de toutes les localités stables (sauf à partir de l'état but) ;
- l'ajout de transitions d'entrée-sortie qui créent un chemin vers le but ayant pour sources des localités transitoires ;
- la suppression des transitions d'entrée-sortie de $\mathcal{S}1$.

6.3.2 Règles de contraintes

Dans cette section, nous décrivons l'algorithme qui construit un contrôleur du système soumis à des règles de contraintes. Afin de simplifier l'algorithme, nous imposons une restriction sur le type de système dont nous pouvons calculer le contrôleur.

Hypothèse 6.1 *Chaque transition d'entrée-sortie permet d'affecter aux variables, toutes les valeurs de leur domaines de la localité destination. C'est-à-dire que chacune des transitions d'entrée-sortie peut amener à tout les états de sa localité destination.*

Cette hypothèse permet de simplifier l'algorithme et notamment le calcul de l'étape 6. Nous discutons dans les perspectives de la thèse, des affaiblissements possibles de cette hypothèse.

L'algorithme synthétisant le contrôleur produit un automate IOSTS contrôleur $\mathcal{C}'$. Ce contrôleur est synthétisé à partir d'une règle de contrainte $\mathcal{R}$ (cette lettre est choisie pour $\mathcal{R}$ule) et d'un automate de comportement $\mathcal{S} = \langle D_\mathcal{S}, \Theta_\mathcal{S}, L_\mathcal{S}, L^0_\mathcal{S}, \Sigma_\mathcal{S}, T_\mathcal{S} \rangle$ avec $D_\mathcal{S} = V_\mathcal{S} \cup P_\mathcal{S}$. De plus, nous avons besoin d'un état de secours représenté par le couple (l^{sec}, Θ^{sec}). Cet état permet d'initialiser un système lorsque le contrôleur démarre et que le système est dans un état illégal.

Nous définissons ci-dessous les éléments nécessaires à la description de l'algorithme.

$L^t_\mathcal{C}$ est l'ensemble des localités transitoires, sous-ensemble des localités du système $\mathcal{C}$. Ces localités transitoire permettent d'indiquer si le dispositif est dans un

état correspondant à un état de $\mathcal{S}$ ou dans un état intermédiaire d'un chemin de réaction destiné à ramener le système dans un état non contraint.

Z est un ensemble de transitions temporaires où la destination doit être dupliquée afin de servir de point de départ pour un chemin de retour vers un état précédent. Les transitions de l'ensemble Z sont traitées par les étapes 5 et 9

$Y \subset T^? \times \underline{L_\mathcal{S}}$ est un ensemble de couples d'une transition de sortie de $t \in T_\mathcal{C}$ et d'une localité stable de $l \in \underline{L_\mathcal{S}}$. Les couples de cet ensemble sont utilisés afin de calculer un chemin qui permet de revenir à l'état précédent. L'élément t dispose d'une localité source l^o inclue à la fois dans $\underline{L_\mathcal{S}}$ et $\underline{L_\mathcal{C}}$ ($l^o \in \underline{L_\mathcal{S}} \cap \underline{L_\mathcal{C}}$) ainsi qu'une localité destination l_d inclue dans $\underline{L_\mathcal{C}}$ mais pas dans $\underline{L_\mathcal{S}}$ ($l_d \in \underline{L_\mathcal{C}} \wedge l_d \notin \underline{L_\mathcal{S}}$). l_d est dupliqué depuis la localité l. La recherche du chemin pour revenir à l'état précédent prend comme point de départ l et comme fin l^o.

$X \subset L^t_\mathcal{C} \cap \underline{L_\mathcal{S}}$ est un sous-ensemble de localités illégales et transitoire de $\mathcal{C}$. Ce sont les localités stables à partir desquelles il doit exister un chemin de secours. Un chemin de secours amène un dispositif dans l'état représenté par le couple (l^{sec}, Θ^{sec}).

W $(\underline{L_\mathcal{S}} \rightarrow 2^{L_\mathcal{C}})$ est une fonction qui, pour une localité donnée l, retourne un sous-ensemble de localités. La fonction W permet de faire le lien entre une localité $l \in L_\mathcal{S}$ et les localités créées dans $\mathcal{C}$ à partir de l par l'algorithme. Les éléments l sont des localités stables et légales du système $\mathcal{S}$. L'élément retour $L'_\mathcal{C} \in 2^{L_\mathcal{C}}$ est un sous-ensemble de l'ensemble des localités de $\mathcal{C}$, $L'_\mathcal{C} \subset L_\mathcal{C}$.
Par exemple, le couple $(l_1, \{l_3, l_5, l_9\})$ indique que la localité l_1 a permis de créer les localités l_3, l_5 et l_9.

Lors de la description de l'algorithme, nous introduisons des fonctions que nous définissons par la suite.

L'algorithme de synthèse crée $\mathcal{C} = \langle D_\mathcal{C}, \Theta_\mathcal{C}, L_\mathcal{C}, L^0_\mathcal{C}, \Sigma_\mathcal{C}, T_\mathcal{C} \rangle$ avec $D_\mathcal{C} = V_\mathcal{C} \cup P_\mathcal{C}$ le miroir du système $\mathcal{C}'$. Il comporte les 10 étapes détaillées ci-dessous.

1. Créer les variables de $\mathcal{C}$.

   Les variables de $\mathcal{C}$ sont les variables de $\mathcal{S}$ plus les variables de sauvegarde de contexte. Ces dernières permettent de revenir à un état précédent lorsque le dispositif rentre dans un état interdit par la contrainte $\mathcal{R}$.

   $$V_\mathcal{C} \supset V_\mathcal{S}$$
   $$\text{et } \forall v \in V_\mathcal{S}\,,\ sauvegardeDe(v) \in V_\mathcal{C}$$

   La fonction $sauvegardeDe$ $(V_\mathcal{S} \rightarrow V_\mathcal{C})$ associe à une variable $v \in \mathcal{S}$, une variable tampon v' qui permet de sauvegarder v. Cette fonction est implémentée par un tableau associatif (*Map* en anglais).

2. Créer les localités stables et légales de $\mathcal{C}$.

   Les localités stables et légales de l'automate de comportement $\mathcal{S}$ appartiennent aux localités stables de $\mathcal{C}$. De plus, elles font partie de l'ensemble des localités initiales.

$$l \in \underline{L_{\mathcal{S}}} \mid Legale(l) \implies l \in \underline{L_{\mathcal{C}}} \wedge l \in L^0_{\mathcal{C}}$$
$$l \in \underline{L_{\mathcal{S}}} \mid Legale(l) \implies l \in W(l)$$

3. Créer les localité stables et illégales de $\mathcal{C}$.

   Les localités stables et illégales de l'automate de comportement $\mathcal{S}$ appartiennent à l'ensemble des localités stables et transitoires de $\mathcal{C}$. De plus, elles doivent être le point de départ d'un chemin pour l'état de secours ; elles appartiennent donc à l'ensemble X. Ces localités font partie de l'ensemble des localités initiales.

$$l \in \underline{L_{\mathcal{S}}} \mid Illegale(l) \implies l \in L^t_C \wedge l \in \underline{L_{\mathcal{C}}} \wedge l \in X \wedge l \in L^0_{\mathcal{C}}$$
$$l \in \underline{L_{\mathcal{S}}} \mid Illegale(l) \implies l \in W(l)$$

4. Diviser les localités stables et partiellement illégales $l \in \underline{L_{\mathcal{S}}}$ afin d'obtenir des localités légales et des localités illégales.

   La division est effectuée de telle manière que l'ensemble des états de la première partie l_1 d'une localité l divisée soient légaux et que les états restants formant la localité l_2 soient illégaux. Le fait que l_2 soit illégale implique qu'il faut créer un chemin à destination de l'état de secours ; l_2 appartient donc à l'ensemble X.

$$\begin{array}{l}\forall l \in \underline{L_{\mathcal{S}}} \mid PIllegale(l), \\ \exists \langle l_1, l_2, T'' \rangle = Div(l, \mathcal{R}) \\ \quad \text{avec } l_1 \in \underline{L_{\mathcal{C}}} \wedge Legal(l_1) \\ \quad \text{et } l_2 \in L^t_C \wedge l_2 \in \underline{L_{\mathcal{C}}} \wedge l_2 \in X \wedge Illegal(l_2) \\ \quad \text{et } \{l_1, l_2\} \subset W(l)\end{array}$$

   Parmi les transitions T'' produites par l'opération, la transition réflexive de l_1 appartient à $\mathcal{C}$

$$t = \langle l_1, a, G, A, l_1 \rangle \in T'' \implies t \in \mathcal{C}$$

   Parmi les transitions T'' produites par l'opération, les transitions de sortie non réflexives allant vers l_1 et provenant d'une localité légale appartiennent à $\mathcal{C}$.

$$\forall t = \langle l, a, G, A, l_1 \rangle \in T'' \mid l \neq l_1 \wedge Legal(l) \wedge a \in \Sigma^! \implies t \in T_{\mathcal{C}}$$

Parmi les transitions T'' produites par l'opération, les transitions de sortie ayant comme destination l_2 et provenant d'une localité légale appartiennent à Z.

$$\forall t = \langle l_n, a, G, A, l_2 \rangle \in T'' \,|\, l \neq l_2 \wedge Legal(l) \wedge a \in \Sigma^! \implies t \in Z$$

Les transitions de l'ensemble Z sont traitées par les étapes 5 et 9. Les transitions ayant pour sources l_2 sont traitées par l'étape 9.

5. Préparer le chemin de retour vers l'état précédent depuis chacune des localités transitives.

   Les localités illégales de $\mathcal{C}$ et $\mathcal{S}$, destinations de transitions de sortie, sont dupliquées afin de préparer un chemin de récupération de l'état précédent (ou chemin de retour). Les destinations des transitions présentes dans l'ensemble Z sont aussi dupliquées afin de créer un chemin de retour. Les transitions de Z ne sont pas dans le système $\mathcal{S}$ ni dans le système $\mathcal{C}$.

   Les transitions de sortie non réflexives ($l^o \neq l^d$) de $\mathcal{S}$ et de Z ayant comme source une localité stable et comme destination une localité stable et illégale forment l'ensemble T'.

   $$t' = \langle l^o, a, G, A, l \rangle \in T' \iff \{l, l^0\} \subset \underline{L_s} \wedge t' \in T_{\mathcal{S}} \cup Z \wedge Illegale(l) \wedge l^0 \neq l \wedge a \in \Sigma^!$$

   Chaque destination des transitions $t' \in T'$ est dupliqué $Dup(l^d) = \langle l_2, T'' \rangle$. La localité ainsi créée appartient à l'ensemble des localités transitoires et à l'ensemble des localités stables $l_2 \in L^t_{\mathcal{C}} \wedge l_2 \in \underline{L_{\mathcal{C}}}$. l^d quant à elle n'appartient pas au système $l^d \notin l_{\mathcal{C}}$.

   $$\begin{array}{l} \forall t' = \langle l^o, a, G, A, l^d \rangle \in T' \\ \exists \langle l_2, T'' \rangle = Dup(l) \\ \quad \text{avec } l_2 \in L^v_{\mathcal{C}} \wedge l_2 \in \underline{L_{\mathcal{C}}} \\ \exists t_2 = \langle l^o, a, G, A', l_2 \rangle \in T_{\mathcal{C}} \\ \quad \text{avec } A' = A \cup SauvegardeContexte(V_{\mathcal{S}}) \\ \quad \text{et } (t_2, l^d) \in Y \\ \quad \text{et } l_2 \in W(l) \end{array}$$

   Les transitions créées par l'opération de duplication ne sont pas reportées dans $\mathcal{C}$. Les transitions de sortie de l^d sont traitées par l'étape 6.

   La fonction $SauvegardeContexte$ $(V \rightarrow A)$ est définie par $f(V) = A'$. L'ensemble V matérialise un ensemble de variables que l'on souhaite sauvegarder. Le résultat A' représente les affectations à réaliser pour sauvegarder le contexte.

$$f(V) = A' = \{(v, sauvegardeDe(v)) \mid v \in V\}$$

6. Créer un chemin de retour vers l'état précédent depuis chacune des localités transitives.

   Pour chacun des couples de $(t = \langle l^a, a, G, A, l^o\rangle, l'^o) \in Y$, tous les chemins simple partant de la localité l^o vers la localité l^a sont recherchés. Les chemins permettant d'accéder à toutes les valeurs du domaine des variables de la localité l^a sont sélectionnés, dupliqués et insérés dans $\mathcal{C}$. Les localités et les transitions dupliquées appartiennent à $\mathcal{C}$.

$$\begin{array}{l}\forall(t = \langle l^a, a, G, A, l^o\rangle, l^o) \in Y \\ \quad ListChemins = trouverChemins(l'^o, l^a) \\ \quad ListChemins' = sélectionnerCheminsC(ListChemins) \\ \quad (L^2, T^2) = créerCheminsDansC(l, ListChemins') \\ \quad L^2 \subset L_{\mathcal{C}} \cap L^t_{\mathcal{C}} \wedge T^2 \subset T_{\mathcal{C}}\end{array}$$

   La fonction $trouverChemins : L_{\mathcal{S}} \times L_{\mathcal{S}} \rightarrow ListeChemins$ $trouverChemins$ $(L_{\mathcal{S}} \times L_{\mathcal{S}} \rightarrow ListeChemins)$ est définie par $trouverChemins(l^o, l^d) = c$. La localité l^o est la source du chemin et la localité l^d représente la localité destination. Cette fonction associe pour des paramètres d'entrée une liste de chemins c. Ce résultat est une liste d'ensembles de transitions non internes $T \subset 2^{T_{\mathcal{S}}}$ permettant d'amener le dispositif à la destination. Le parcours commence par les transitions d'entrée ayant comme origine l^o. L'algorithme utilise un parcours de toutes les transitions et donne en retour tous les chemins simples possibles pour aller de l^o à l^d.

   La fonction $sélectionnerChemins$: $ListeChemins \rightarrow ListeChemins$ est définie par $sélectionnerChemins(l) = l'$. Le paramètre l représente une liste de chemins allants de la localité l^s vers la localité l^d. Cette fonction associe pour un paramètre une liste de chemins l'. Le résultat l' est l'ensemble permettant d'atteindre le nombre maximum d'états de l^d depuis l^s. En raison de l'hypothèse 6.1, l'implantation de cette fonction retourne une liste composée uniquement du chemin le plus court.

   La fonction $créerCheminsDansC$ $(L \times ListeChemins \rightarrow 2^L \times 2^T)$ est définie par $f(l, Lc) = (L', T')$. La localité l représente une liste de chemins allants de la localité l^s vers la localité l^d. Cette fonction appelle, pour chaque chemin de l, une fonction $créerCheminDansC$.

$$\begin{array}{l}f(l) = (L, T) \mid \forall c \in Lc \\ \quad (L', T') = créerCheminsDans(l, c) \\ \quad \text{avec } L' \subset L \text{ et } T' \subset T\end{array}$$

La fonction récursive *créerCheminsDans*$\mathcal{C}$ ($L \times 2^{Ls} \rightarrow 2^L \times 2^T$) est définie par $g(l, c) = (L', T')$. *créerCheminsDans*$\mathcal{C}$ crée les transitions du chemin et les localités intermédiaires compatibles avec $\mathcal{C}$, le résultat doit être intégré dans le système $\mathcal{C}$. Elle adapte les paramètres des transitions d'entrée afin de récupérer l'ancien état en appelant la fonction *adapterGarde*. Le calcul de $g(l, c)$ est donné par l'algorithme suivant :

$$g(l,c) = \left\{ \begin{array}{l} \text{Si } c = [\, t^h \,|\, [] \,] \text{ avec } t^h = \langle l^a, a, G, Al^b \rangle \\ \quad t^d = \langle l, aGAl^b \rangle \\ \quad \text{résultat}(\emptyset, \{t^d\}) \\ \\ \text{Sinon si } c = [\, t^h \,|\, C^q \,] \text{ avec } t^h = \langle l^a, a, G, Al^b \rangle \text{ et } C^q \neq [] \\ \quad \exists (l^d, T^d) = Dup(l) \\ \qquad \text{avec } l^d \in W(l) \\ \quad \exists (L', T') = g(l^d, C^q) \\ \quad \exists t^d = \langle l, a, G', A, l^d \rangle \text{ avec } G' = adapterGarde(t^h) \\ \quad \text{résultat}(L' \cup \{l^d\}, T' \cup \{t^d\}) \\ \text{Fin si} \end{array} \right.$$

La fonction itérative *adapterGarde* ($T_{\mathcal{S}} \rightarrow G$) définie par $f(t) = g$ qui pour une transition donnée, retourne la garde qui permet de connaître les valeurs à affecter aux paramètres des transitions d'entrée. *créerCheminsDans*$\mathcal{C}$ crée les transitions du chemin et les localités intermédiaires compatibles avec $\mathcal{C}$, le résultat doit être intégré dans le système $\mathcal{C}$. Cette dernière fonction adapte les paramètres des transitions d'entrée afin de récupérer l'ancien état. Nous avons donc l'algorithme :

$$f(t) = \left\{ \begin{array}{l} t = \langle l^a, a, G, A, l^b \rangle \\ \text{Si } a \in \Sigma^? \\ \quad \exists t^b = \langle l^b, a', G', A', l^b \rangle \\ \quad G^{res} := G \\ \quad \forall (varTmp, valParm) \in A \\ \qquad \exists (var, varTmp) \in A' \\ \qquad varSav = sauvegardeDe(var) \\ \qquad G^{res} := G^{res} \wedge \text{« } var = varSav \text{ »} \\ \quad \text{résultat } G^{res} \\ \text{Sinon} \\ \quad \text{résultat } G \\ \text{Fin si} \end{array} \right.$$

7. Créer un chemin de secours pour chacune des localités stables et illégales.

Pour chacune des localités de X, un chemin de secours est calculé. Le calcul produit une liste ordonnée de transitions.

$$\begin{array}{l}\forall l \in X \\ \quad l' = créerPar(l) \\ \quad T' = trouverCheminJusquAEtat(l', (l^{sec}, \Theta^{sec})) \\ \quad (L'', T'') = dupliquerCheminVersEtat\mathcal{C}(l, T') \\ \quad L'' \subset L^t_{\mathcal{C}} \wedge T'' \subset T_{\mathcal{C}}\end{array}$$

La fonction $créerPar$ $(L_{\mathcal{S}} \rightarrow L_{\mathcal{C}})$ définie par $créerPar(l) = l'$ donne pour la localité $l \in L_{\mathcal{C}}$ la localité $l'_{\mathcal{S}}$ qui a permis de la créée.

La fonction $trouverCheminJusquAEtat$ $(L_{\mathcal{S}} \times (L_{\mathcal{S}}, \Theta) \rightarrow 2^{T_{\mathcal{S}}})$ est définie en section 6.3.1.

La fonction $dupliquerCheminVersEtat$ $(L_{\mathcal{S}} \times 2^{T_{\mathcal{S}}} \rightarrow 2^L \times 2^T)$ est définie en section 6.3.1.

8. Ajout des transitions de sortie entre les localités stables, légales et non transitoires.

   Pour toutes les localités stables et légales, ajouter les transitions de sortie de $\mathcal{S}$ à $\mathcal{C}$.

$$\begin{array}{l}\forall t = \langle l^o, a, G, A, l^d \rangle \mid l^o \in \underline{L_{\mathcal{C}}} \ \wedge \ Legal(l^o) \ \wedge \ l^o \notin L^v_{\mathcal{C}} \wedge \\ \quad a \in \Sigma^? \implies t \in T_{\mathcal{C}}\end{array}$$

9. Ajout des transitions de sortie entre les localités stables et transitoires et les localités initiales de $\mathcal{C}$. Ces transitions permettent de sortir des localités stables et transitoires lorsqu'une notification parvient et n'est pas attendue.

   Pour chaque transition de sortie $t \in \mathcal{S}$ ayant comme source une localité stable et légale l^o et comme destination l^d et pour toutes les localités stables et transitoires l^v créées à partir de l^s, si l^d est légale reporter une copie de la transition t entre l^v et les l^d. Sinon l^d est partiellement illégale. Dans ce cas, trouver les deux localités l^{da}, l^{db} créées à partir de l^d et copier la transition t entre l^v et les l^{da} et entre l^v et les l^{db}. Les gardes des transitions créées sont

adaptées en fonction du domaine des localités de destination.

$$
\begin{array}{l}
\forall t = \langle l^o, a, G, A, l^d \rangle \in T_{\mathcal{S}} \mid \underline{L_{\mathcal{C}}} \ \wedge \ Legale(l^o) \ \wedge \ a \in \Sigma^? \\
\quad \forall l^v \in W' \mid W' = W(l^o) \wedge l^v \in L^v_{\mathcal{C}} \\
\qquad \exists\, l^{da} \in W'' \mid W'' \in W(l^d) \ \wedge \ l^{da} \in \underline{L_{\mathcal{C}}} \wedge Legale(l^{da}) \\
\qquad Legale(l^d) \implies \\
\qquad\quad \exists t' = \langle l^v, a, G, A, l^{da} \rangle \in T_{\mathcal{C}} \\
\qquad PIllegale(l^d) \implies \\
\qquad\quad \exists l^{db} \in W' \mid (l^d, W') \in W \wedge l^{db} \in \underline{L_{\mathcal{C}}} \wedge Illegale(l^{db}) \\
\qquad\quad \exists t' = \langle l^v, a, G', A, l^{da} \rangle \in T_{\mathcal{C}} \mid G' = G \wedge adapteGarde(t, \mathcal{R}) \\
\qquad\quad \exists t'' = \langle l^v, a, G'', A, l^{db} \rangle \in T_{\mathcal{C}} \mid G'' = G \wedge \neg adapteGarde(t, \mathcal{R})
\end{array}
$$

Soit G l'ensemble des expressions booléennes. La fonction itérative $adapteGarde$: $T_{\mathcal{S}} \times G \rightarrow G$ est définie par $f(t, c) = g$. Cette fonction adapte la garde de la transition t en fonction de la condition c. c est une expression booléenne sur les variables de $\mathcal{S} \cap \mathcal{C}$. Dans l'implantation, nous la traitons comme une chaîne de caractères où nous remplaçons le nom des variables par le nom du paramètre. Nous avons l'algorithme ci-dessous :

$$
f(t,c) = \begin{cases}
\text{résultat } true & \text{Si } PIllegale(l) \\
\\
G' := c & \\
t = \langle l^a, a, G, A, l^b \rangle & \\
\forall (var, val) \in A & \text{Sinon} \\
\qquad \text{Remplacer dans G' la chaine} & \\
\quad G' := \ \text{de caractères de valeur var} & \\
\qquad \text{par la valeur de val.} & \\
\text{résultat } G' &
\end{cases}
$$

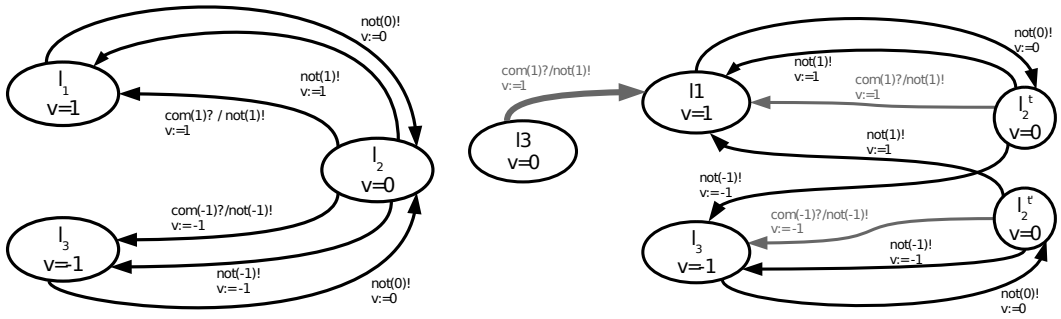
10. Inverser le miroir pour obtenir l'automate contrôleur.

    L'opération $Miroir$ est utilisée à la fin de l'algorithme afin d'inverser les actions d'entrée et les actions de sortie.

$$\mathcal{C}' = Miroir(\mathcal{C})$$

En considération de ce qui précède, tous les états d'une localité de l'automate du contrôleur satisfont la contrainte ou ne la satisfont pas. Les localités dont les états satisfont (respectivement, ne satisfont pas) la contrainte sont nommées localités non contraintes (localités contraintes).

Par exemple, la figure 6.6 illustre la synthèse d'un contrôleur issu d'une règle de contrainte et d'un dispositif fictif. La règle de contraînte est $v \neq 0$ et l'état de



(a) Automate de comportement d'un système fictif

(b) *Miroir* de l'automate du contrôleur

FIGURE 6.6 – Synthèse d'un contrôleur d'un dispositif fictif avec une contrainte $v \neq 0$

secours est $\langle l1,\ v = 1\rangle$. Le dispositif contient trois états représentés par les trois localités $l1$, $l2$, $l3$. Le domaine de la variable pour la localité $l1$ est $v = 1$, $l2$ est $v = 0$ et $l3$ est $v = -1$. Dans la figure, la dernière ligne de l'étiquette des localité est le domaine de cette localité. La figure 6.6(b) présente le contrôleur synthétisé. Les différences entre le contrôleur (6.6(b)) et le dispositif (6.6(a)) sont :

- l'existence de deux localités l_2 *et* l'_2 remplaçant la localité l_2 ;
- deux transitions d'entrée-sortie à destination de $l1$ *et* $L3$;
- un chemin de secours depuis la localité $l3$ jusqu'à l'état de secours.

6.4 Synthèse

Dans ce chapitre, nous utilisons le terme contrôleur en tant que système de contrôle extérieur. En effet, le terme contrôleur ou superviseur est utilisé dans la littérature pour un système qui permet de restreindre le comportement d'un système. Nous utilisons ce terme pour indiquer un système permettant d'envoyer des commandes de façon automatique à un système afin que son état corresponde à des règles de conduite. Pour résumer, le contrôleur est une télécommande automatique programmée par des règles.

Ce chapitre décrit les algorithmes de synthèse de contrôleurs pour des dispositifs de la maison que nous avons développés. La synthèse se base sur un système à commander et sur une règle. L'algorithme de synthèse dépend du type de règle. Cette dernière peut être :

- une règle de changement par action sur évènement,
- une règle de changement par but sur évènement,

– une contrainte.

Le chapitre introduit les opérateurs et les termes qui portent sur les automates IOSTS nécessaires à la construction du contrôleur. Par exemple, nous avons introduit l'opération de division de localité et le terme de localité illégale.

Un contrôleur est applicable à un seul système. Lorsqu'un contrôleur doit être synthétisé pour un ensemble de dispositifs, l'opérateur somme de comportements est utilisé entre les automates des dispositifs afin de créer un système unique. Ce système est le comportement global des dispositifs évoluant indépendamment les uns des autres.

Sommaire

Chapitre 7

Mise en œuvre

Les chapitres précédents de ce mémoire formalisent un modèle de comportement des objets communicants sous forme d'automates d'entrées-sorties. Ils décrivent aussi les automates contrôleurs. Ce sont des automates qui permettent de piloter les objets communicants. Ces automates sont synthétisés à partir du comportement des systèmes d'un environnement et de règles de comportement.

Dans un premier temps, ce chapitre indique comment est composée la chaîne de traitement permettant de produire les automates contrôleurs et comment ces derniers sont utilisés. Ensuite, il présente la mise en œuvre du calcul de contrôleurs de manière automatique sur un exemple.

7.1 Dynamique de fonctionnement

La figure 7.1 est une vue des étapes et dépendances permettant de créer et d'utiliser les automates contrôleurs. La figure contient différents symboles :

- Les rectangles représentent les opérations. Les flèches pleines à destination d'une opération correspondent aux éléments dont dépend l'opération. Les flèches ayant pour source un rectangle sont la production de l'opération.
- Les cylindres représentent des descriptions ne changeant pas ou peu. Le libellé interne indique ce que représente le cylindre. Lorsque ces descriptions changent, il faut alors recalculer les sorties des opérations dont les descriptions dépendent.

– Le papyrus représente une description calculée. Il est créé par l'opération de construction de l'automate contrôleur.
– Le nuage correspond aux systèmes de l'environnement. Ces systèmes envoient des notifications à l'application de gestion des envois de commandes et reçoivent des commandes de ce dernier. Ces échanges sont représentés par les flèches en pointillés.

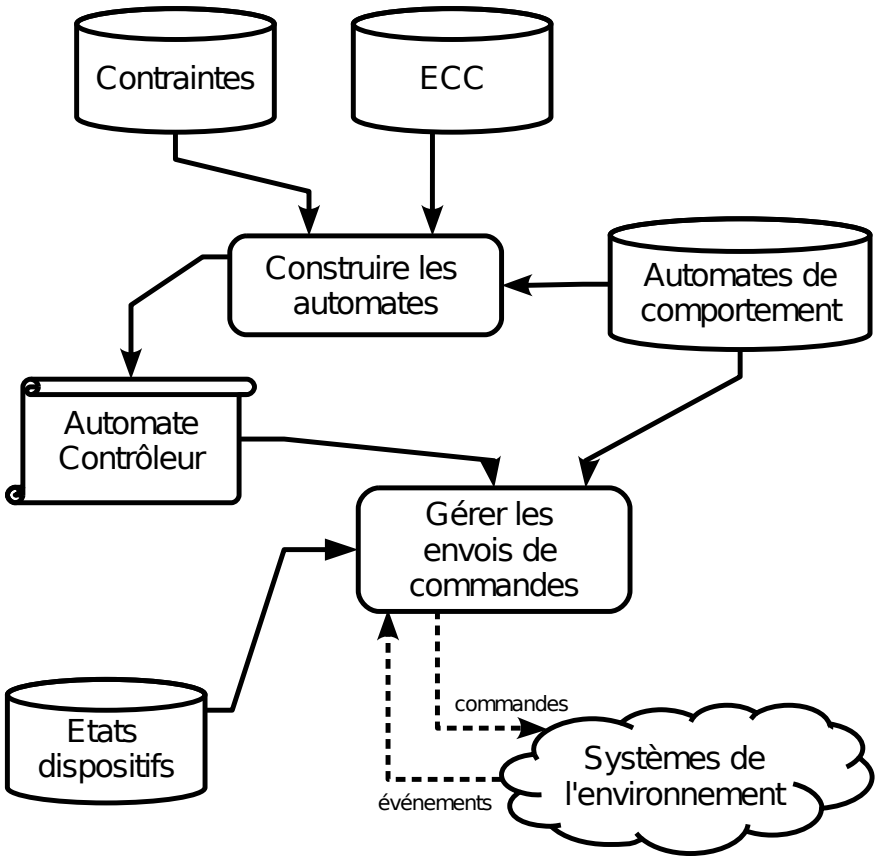


FIGURE 7.1 – Relation entre automates, ECC et contraintes

L'opération *construire les automates* génère des automates contrôleurs. Cette opération a besoin des automates de comportement des dispositifs, des contraintes et des règles à appliquer sur le système. Lorsqu'une règle porte sur plusieurs systèmes, l'opération applique l'opérateur de composition *somme de comportements* ($\oplus$) entre les différents systèmes afin d'obtenir un automate unique. Puis, l'opération utilise les règles ECC ou les contraintes pour créer l'automate du contrôleur.

L'opération *gérer les envois de commandes* permet de réaliser des actions sur le système. Celle-ci s'occupe d'envoyer les ordres aux dispositifs dans le but de les

mettre en position. Cette opération a besoin des automates calculés par l'opération précédente afin de connaître les ordres à envoyer. De plus, cette opération écoute les actions de sortie du système commandé afin de réagir lorsqu'un évènement décrit par une règle ECC intervient. Enfin, les états des systèmes sont nécessaires pour connaître la localité courante du contrôleur

Il convient de rappeler que les états de l'automate du contrôleur correspondent à une sous partie des états des systèmes de l'environnement. Lorsque la description de l'*état des dispositifs* change, l'opération *gérer les envois de commandes* doit retrouver l'état courant du système dans ses automates contrôleurs. Puis, il calcule les commandes à envoyer et les envoie.

7.2 Présentation de l'exemple

Ce chapitre traite du calcul de contrôleur permettant de piloter des objets communicants. Ils sont construits à partir de différentes règles. Les objets communicants permettant de créer l'exemple sont une fenêtre oscillo-battante et un interrupteur deux états.

7.2.1 Fenêtre oscillo-battante

La fenêtre dispose des trois positions : fermée, ouverte en soufflet et ouverte à la française. Les localités associées aux positions sont nommées respectivement : *fermée*, *souverte* et *fouverte*. Cette fenêtre est particulière. En effet, elle s'ouvre dans l'une ou l'autre position que lorsqu'elle est fermée. Dans le cas où son état courant est dans la localité *souverte* (respectivement *fouverte*) elle ne peut pas s'ouvrir dans la localité *fouverte* (respectivement *souverte*). Seules les actions de fermeture et d'ouverture dans la même position sont autorisées lorsque la fenêtre est ouverte. En revanche, lorsque le dispositif est fermé, l'action d'ouverture est autorisée pour les deux positions.

Nous intégrons cette particularité afin de compliquer l'exemple. De cette manière, le passage d'un état ouvert à la française à un état ouvert en soufflet n'est pas trivial et nécessite de fermer la fenêtre avant de l'ouvrir. Ce type de dispositif n'existe pas. En réalité, une porte oscillo-battante complètement télécommandée pourrait passer de l'une à l'autre localité ouverte sans aucun problème.

Ce comportement est reporté par la figure 7.2. La figure utilise le raccourci de notation ; il y a en réalité neuf localités, six localités virtuelles et trois localités stables. Les localités virtuelles sont incluses dans les transitions d'entrée-sortie telles que *ouvrir(p) ?/not(p) !* et *fermer(p) ?/not(p) !*. Les trois localités stables sont les localités initiales. La condition initiale de ce système est que la variable s

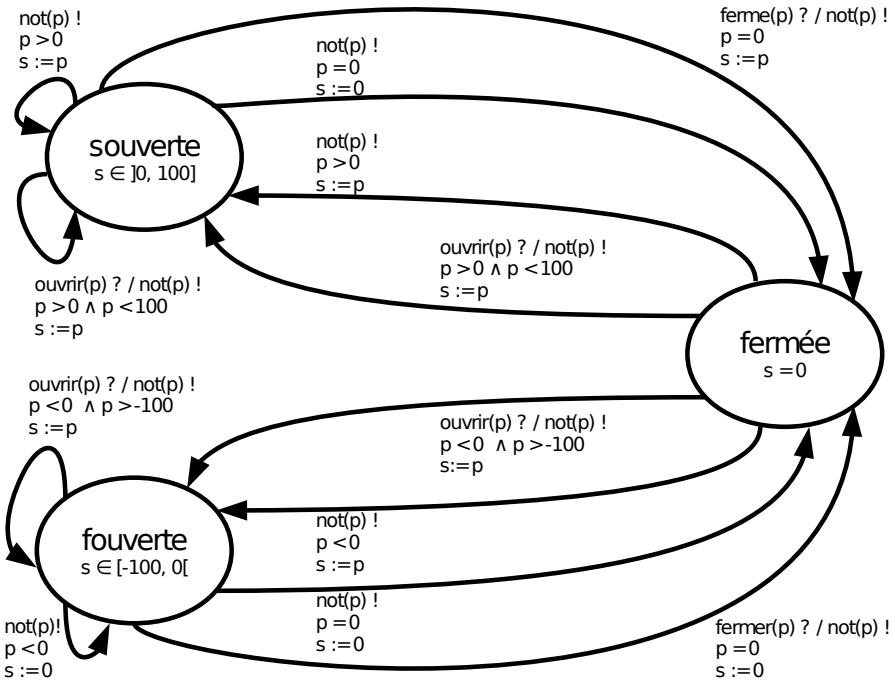


FIGURE 7.2 – Automate de comportement d'une fenêtre oscillo-battante

soit initialisée avec une valeur entre -100 et 100. La figure A.1 en annexe page 146 donne l'automate complet utilisé pour calculer les contrôleurs.

Pour la suite, l'automate de la fenêtre sera $\mathcal{S}_f = \langle D_f, \Theta_f, L_f, \underline{L_f}, \Sigma_f, T_f \rangle$. L'ensemble des données contient la variable d'ouverture $D_f = \{s\}$. Le domaine de la variable s est compris entre -100 et 100. Une valeur négative indique une ouverture à la française alors qu'une valeur positive indique une ouverture en soufflet. La valeur absolue de la variable correspond au pourcentage d'ouverture de la fenêtre. Une valeur nulle représente l'état fermé.

Pour faciliter la compréhension, le domaine de la variable s est indiqué sous le nom de chaque localité stable.

7.2.2 Interrupteur deux états

L'interrupteur possède deux états correspondant à la position physique de sa partie mobile. En électricité, l'interrupteur est un appareil permettant de fermer ou ouvrir un circuit électrique. Dans le cas d'un interrupteur communicant, celui-ci est responsable de l'émission de notifications lors des changements de position. L'état de l'interrupteur n'est donc pas *fermé* ou *ouvert*, mais celui de la position physique de la partie mobile. Nous appelons ces deux états : *état1* et *état2*.

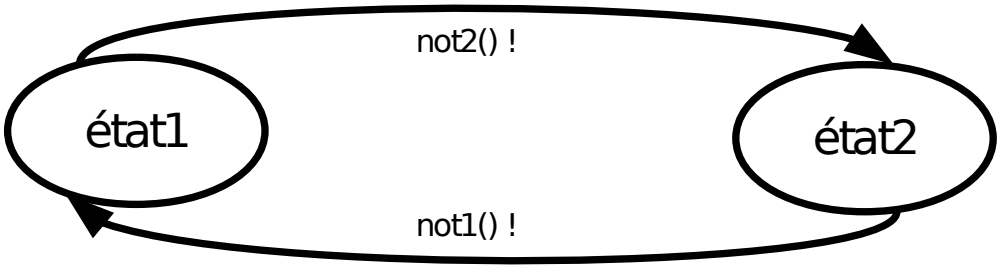


FIGURE 7.3 – Automate de comportement d'un bouton deux états

L'automate de comportement de l'interrupteur communicant est reporté sur la figure 7.3. Par la suite, l'automate de l'interrupteur sera $\mathcal{S}_i = \langle \emptyset, \emptyset, L_i, \underline{L_i}, \Sigma_i, T_i \rangle$ $\mathcal{S}_i$ est composé de deux localités et deux transitions de sortie. Il n'existe pas de variable ni de paramètre dans cet automate, l'ensemble des données est vide. Les localités sont en réalité des états, d'où les noms *état1* et *état2* de ces localités.

L'interrupteur émet les notifications $not1$ lorsqu'il passe de l'état2 à l'état1 et $not2$ lorsqu'il passe de l'état1 à l'état2.

L'interrupteur est un organe de commande, il ne fait qu'émettre et ne peut pas être commandé. Son but est de piloter à distance l'activation ou la désactivation d'un autre système. Dans le cadre de cette thèse, nous détournons le but du bouton afin de nous en servir comme organe capteur de l'intention d'un usager. En effet, le bouton s'adresse à la partie *gestion des envois de commande* de la figure 7.1. C'est cette partie qui a la connaissance des actions à entreprendre.

7.3 Calcul effectif des ECC

Dans cette section, nous allons exécuter l'algorithme de calcul des règles ECC.

7.3.1 Calcul de règle ECA

Afin d'illustrer les *ECA*, nous appliquons la règle suivante : $\langle not1()!, (s > 50 \vee s < -50), ferme()? \rangle$. La notification $not1()$ appartient au bouton $\mathcal{S}_i$ et l'action $ferme()$ appartient à la fenêtre $\mathcal{S}_f$. Cette règle permet de fermer la fenêtre lorsque le bouton passe de l'*état2* à l'*état1* et que la fenêtre est ouverte à plus de 50% en soufflet ou à la française.

La figure 7.4 présente le miroir du contrôleur du dispositif, c'est le comportement désiré. Lorsque la localité courante est source d'une transition d'entrée, le contrôleur doit déclencher l'action de cette transition. Par exemple, les transitions étiquetées avec l'action $ferme()?$ doivent être déclenchées lorsque la localité

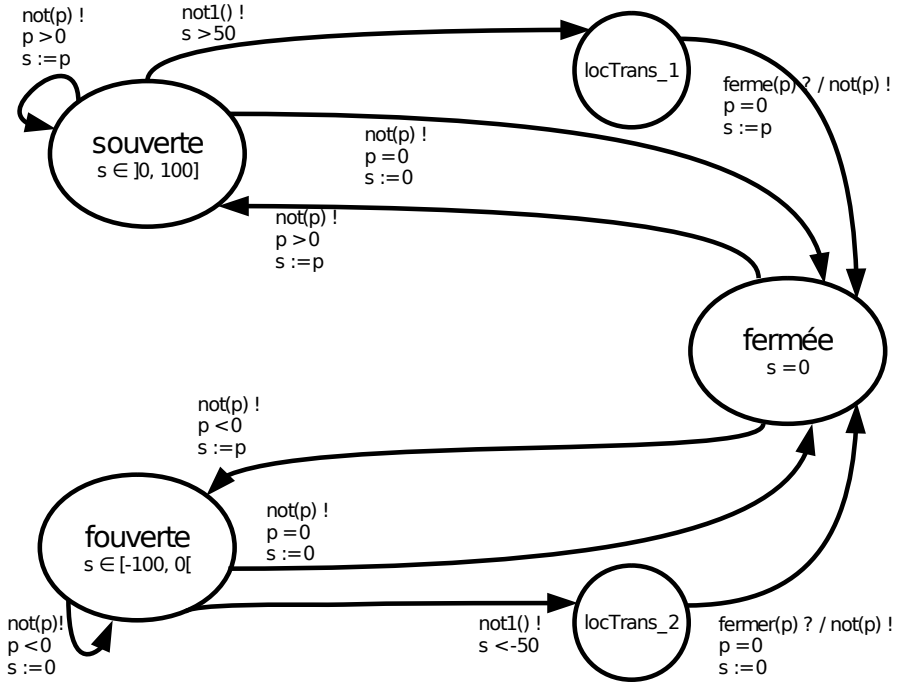


FIGURE 7.4 – Miroir du contrôleur ECA $\langle not1()!, (s > 50 \vee s < -50), ferme()?\rangle$

courante est une des localités transitoires $locTrans\_1$ ou $locTrans\_2$. Ensuite, le dispositif réagit à la commande et émet un retour sous forme d'une notification.

Remarquons que l'action $not1()!$ n'est pas paramétrée. Si nous avions utilisé une action paramétrée, nous aurions indiqué la valeur du ou des paramètres dans la règle.

L'automate possède des localités qui sont l'origine de transitions d'entrée ; elles ne sont accessibles qu'après le déclenchement de l'action de sortie $not1()!$. Ce sont les localités transitoires $locTrans\_1$ et $locTrans\_2$. Elles ne correspondent pas à un état différent de la localité stable précédente, mais marquent le besoin d'envoyer une commande.

Un contrôleur calculé avec une autre règle est présenté en annexe A.2, page 147. Nous pouvons remarquer que le calcul a créé quatre chemins d'entrée-sortie de détection de l'évènement et de réaction. Ce sont ceux disposant des localités « $transi\_i$ » avec i entre 1 et 4.

Ces chemins de réaction ne sont pas tous utiles. La condition de la transition initiale du chemin indique que le domaine de la variable doit être entre 0 et 50. La variable possède un tel domaine dans la localité $SOuverte$. C'est pourquoi, seul le chemin disposant de la localité $transi_3$ est utile.

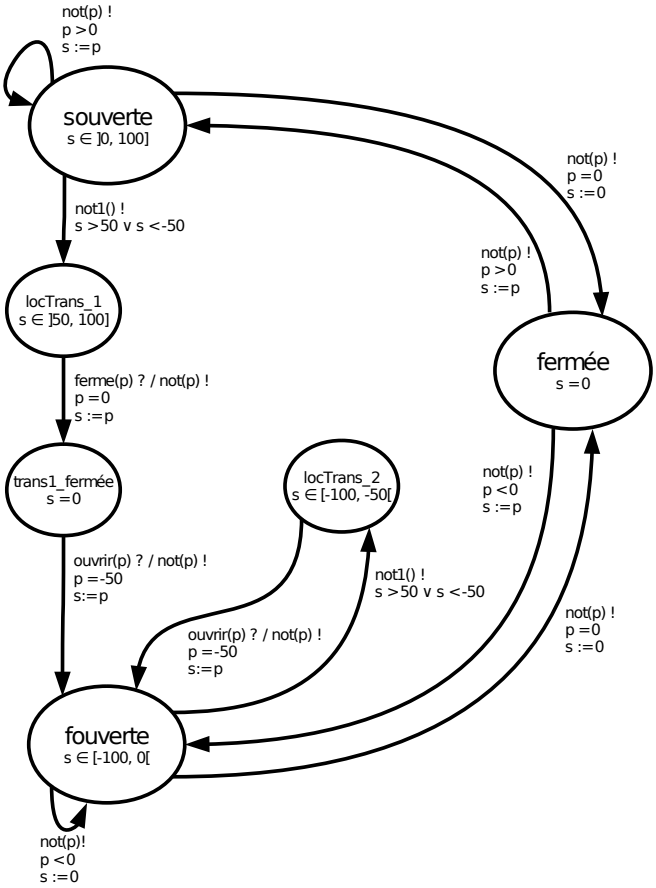


FIGURE 7.5 – Miroir du contrôleur ECB $\langle not1()!, (s > 50 \vee s < -50), \langle FOuverte, s == -50\rangle\rangle$

Nous pourrions améliorer l'algorithme en supprimant les chemins dont les conditions de la transition initiale ne peuvent pas être remplies à partir des localités stables. Cet algorithme supprimerait alors les chemins disposant des localités $trans\_1, trans\_2$ et $trans\_4$.

7.3.2 Calcul de règle ECB

Afin d'illustrer les *ECB*, nous appliquons la règle suivante $\langle not1()!, (s > 50 \vee s < -50), \langle FOuverte, s = -50\rangle\rangle$. La notification $not1()$ appartient au bouton $\mathcal{S}_i$ et la localité $FOuverte$ appartient à la fenêtre $\mathcal{S}_f$. Cette règle permet d'ouvrir la fenêtre à la française à 50% lorsque le bouton passe de l'*état2* à l'*état1* et que la fenêtre et ouverte à plus de 50% en soufflet ou à la française.

La figure 7.5 présente le miroir du contrôleur du dispositif, c'est le comporte-

ment désiré. Lorsque la localité courante est source d'une transition d'entrée, le contrôleur doit déclencher l'action de cette transition. Par exemple, les transitions étiquetées avec l'action $ferme()?$ ($ouvrir(p)?$) doivent être déclenchées lorsque la transition courante est l'une des transitions transitoires $locTrans\_1$ ($locTrans\_2$ ou $Virt1\_fermée$). Dans le cas de la transition étiquetée avec l'action $ouvrir(p)?$, le paramètre p dispose d'une contrainte, le contrôleur doit alors satisfaire la contrainte et donner une valeur au paramètre. Ici la valeur du paramètre doit être -50. Ensuite, le dispositif réagit à la commande et émet un retour sous forme d'une notification.

L'automate possède des localités qui sont l'origine de transitions d'entrée ; elles ne sont accessibles qu'après le déclenchement de l'action de sortie $not1()!$. Ce sont les localités transitoires $locTrans\_1$ et $locTrans\_2$. Elles ne correspondent pas à un état différent de la localité stable précédente, mais marquent le besoin d'envoyer une commande. Le domaine de $locTrans_1$ et $locTrans_2$ est différent de celui des localités qui les précèdent. En effet, les transitions de notifications permettant d'atteindre les localités transitoires disposent de gardes qui restreignent le domaine.

Un contrôleur calculé avec une autre règle est présenté en annexe A.3, page 148. Nous pouvons remarquer que le calcul a créé trois chemins de détection de l'évènement et de réaction. Ce sont ceux disposant des localités $transi\_i$ avec i entre 1 et 3.

Ces chemins de réaction ne sont pas tous utiles. La condition de la transition initiale du chemin indique que le domaine de la variable doit être entre supérieur à 50 ou inférieur à -50. La variable possède un tel domaine dans les localités $SOuverte$ et $FOuverte$. En revanche, ce n'est pas le cas de la localité $Fermée$, le domaine de la variable val rend la satisfaction de la garde impossible. C'est pourquoi, le chemin disposant de la localité $transi\_2$ est inutile.

Nous pourrions améliorer l'algorithme en supprimant le chemin dont les conditions de la transition initiale ne peuvent pas être remplies à partir des localités stables. Cet algorithme supprimerait alors les chemins disposant des localités $trans\_2$ et $virtual\_2\_uFermeeFOuverte$.

7.4 Calcul effectif des contraintes

Afin d'illustrer les *règles de contraintes*, nous appliquons la règle suivante $\langle (s > 0 \vee s < -50), \langle souverte, s = 65 \rangle \rangle$ à la fenêtre $\mathcal{S}_f$. Cette règle informe le contrôleur qu'il doit garder la fenêtre à un niveau d'ouverture de plus de 50% à la française et de plus de 0% en soufflet. Lors de l'initialisation, la fenêtre peut être dans un état interdit. L'état de secours $\langle souverte, s = 65 \rangle$ indique au contrôleur comment commander la fenêtre lorsqu'elle est dans un état initial interdit.

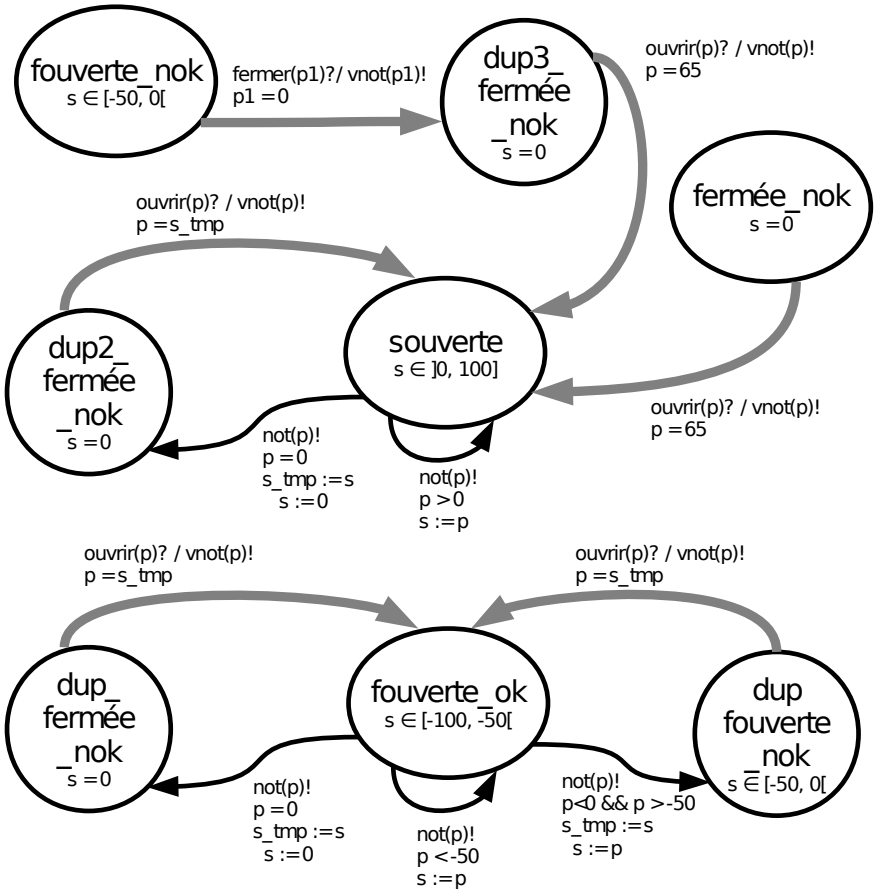


FIGURE 7.6 – Miroir du contrôleur de la contrainte $\langle (s > 0 \vee s < -50), \langle souverte, s = 65 \rangle \rangle$

Cette contrainte implique lors du calcul du miroir du contrôleur $\mathcal{C}$ que :

- l'état $fermee$ est illégal ;
- l'état $souverte$ est légal ;
- l'état $fouverte$ est partiellement illégal.

Le système $\mathcal{C}$ correspond au miroir de l'automate du contrôleur du dispositif sous contrainte. Comme il est impossible d'interdire au système d'exécuter une commande, $\mathcal{C}$ décrit le comportement désiré. La figure 7.6 présente le miroir de $\mathcal{C}$ sans les transitions de sortie depuis les localités légales et transitives. C'est-à-dire que l'algorithme de création de $\mathcal{C}$ est appliqué sans les étapes 9 et 10 de l'algorithme de la section 6.3.2. De la même manière que les autres figures de ce chapitre, la dernière ligne de l'étiquette des localités représente le domaine de la variable du taux d'ouverture. La même figure est disponible en annexe A.4 sans la

simplification des transitions d'entrée-sortie, et la figure A.5 en annexe présente $\mathcal{C}$ avec l'étape 9 .

Les localités initiales sont *fermée_nok*, *fouverte_ok*, *fouverte_nok* et *souverte*. Les transitions d'entrée-sortie sont représentées en gris et d'une épaisseur plus importante que les autres flèches. Les localités transitoires ne sont pas toutes représentées dans la figure ; seules les localités transitoires stables y figurent. Les localités virtuelles sont sous-entendues par les transitions d'entrée-sortie.

L'algorithme de calcul du contrôleur $\mathcal{C}$ permettent de réaliser à l'étape :

- 2, l'introduction de la localité *souverte* ;
- 3, l'introduction de la localité *fermée*, renommé ici en *fermée_nok* ;
- 4, la division de la localité *fouverte* en deux localités *fouverte_ok* et *fouverte_nok* ;
- 5, la duplication des localités *fermée* et *fouverte_nok* pour produire les localités *dup_fermée_nok*, *dup2_fermée_nok* et *dup_fourverte_nok*, ainsi que la création de la première étape des chemins permettant de revenir à un état précédent ; ce sont les transitions de sortie ayant comme destination les localités illégales et transitoires *dup_fermée_nok*, *dup2_fermée_nok* et *dup_fourverte_nok* ;
- 6, de créer le reste du chemin permettant de revenir à un état précédent ; ce sont les transitions d'entrée-sortie ayant comme source les localités illégales et transitoires citées dans le point précédent ;
- 7, de créer les chemins allant vers l'état de secours ; ce sont les chemins ayant comme source les localités *fouverte_nok*, *dup3_fermée_nok* et *fermée_nok* ;
- 9, de créer les relations entre les localités stables et les localités légales et non transitoires ; les transitions de ce type ne sont pas présentes dans la figure 7.6 pour des raisons de lisibilités, en revanche elles sont présentes dans la figure A.5 en annexe page 150 ;
- 10, le miroir de $\mathcal{C}$ pour obtenir un automate de contrôleur.

Remarquons que les actions d'entrée des chemins permettant de revenir à un état précédent et que les chemins allant vers l'état de secours sont paramétrées lorsque c'est nécessaire. L'affectation des valeurs aux paramètres est effectuée par les étapes 6 et 7.

7.5 Implantation

Cette section décrit le logiciel Ibyla (pour I Believe in You Little Application). Ibyla est une partie des développements que j'ai réalisés au cours de ma thèse.

Il correspond à l'implantation des algorithmes vus dans le chapitre précédent. Les programmes fonctionnent sous une machine virtuelle java (JVM) version 1.6. Cependant, ils devraient être fonctionnels sous une JVM 1.5 avec l'ajout d'un module remplaçant la partie JSR-223 de la JVM 1.6. Nous reviendrons sur ce dernier point dans la suite de cette section.

La réalisation et l'exécution de l'application Ibyla font appel à différentes technologies. Certaines ne sont pas présentées par la suite.

- log4j et commons-logging pour les traces applicatives ;
- JUnit 4 pour les tests unitaires ;
- maven pour la gestion de la construction et le packaging de l'application ;
- eclipse pour écrire le code Java et intelliJ IDEA pour écrire le code Scala.

Bien entendu, Ibyla est une preuve de concept. Il ne peut être utilisé tel quel en production. Les améliorations devront porter sur les optimisations des algorithmes proposés dans les perspectives.

Après une présentation de l'architecture globale du logiciel, cette section expose le modèle objet d'un automate IOSTS. Nous présentons ensuite les technologies permettant de créer les modules non fonctionnels. Enfin, nous discuterons d'un choix technologique pour l'implantation des algorithmes.

7.5.1 Architecture du logiciel

Ibyla est mis en place selon une architecture divisée en modules. Cette approche permet de découper l'application en fonction des différentes préoccupations. La figure 7.7 est une modélisation des modules de l'application. Elle présente un diagramme de *packages* (modules) dans le format de la norme UML. Le nom des *packages* permet de comprendre le périmètre fonctionnel de ces modules.

L'approche est de type trois tiers avec :

- la couche présentation correspondant au module *Application* ;
- la couche logique correspondant à *Outils IOSTS* ;
- la couche données correspondant à *Aide IOSTS*.

Les modules non cités sont une aide pour ces trois couches.

Les sous-modules inclus dans le *package Application* orchestrent chacun un programme. Ce sont des programmes lignes de commandes qui permettent à un utilisateur de se servir de l'application. Les trois programmes sont : *Iosts Explorateur*, *Outils IOSTS* et *Calcul de règles*.

Iosts Explorateur permet de parcourir un automate IOSTS. Le programme sollicite l'utilisateur afin de connaître quelles sont les actions à déclencher et avec

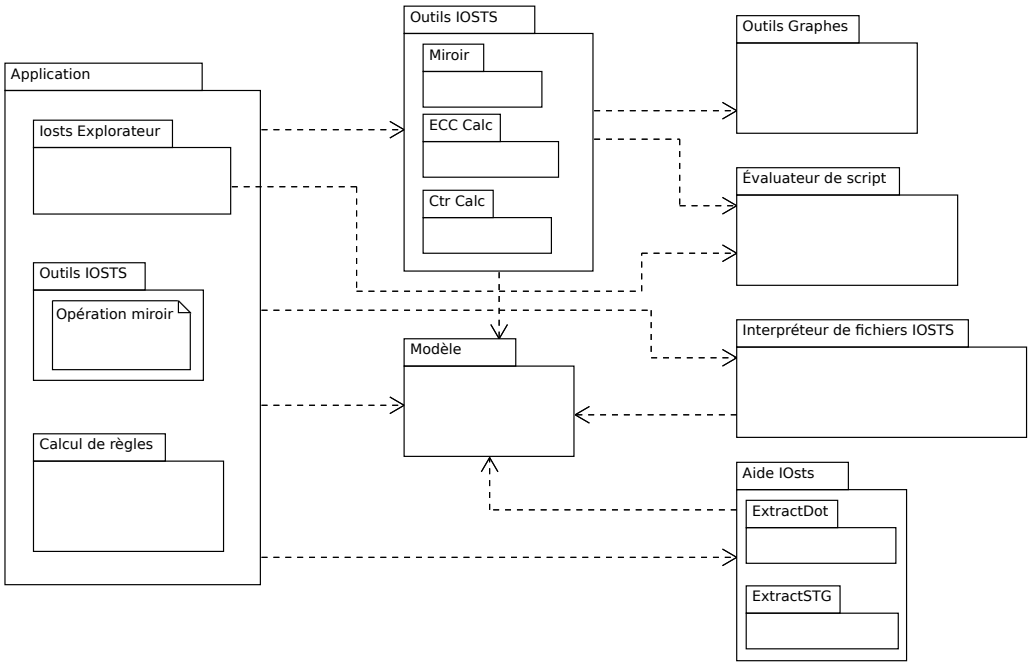


FIGURE 7.7 – Représentation des *packages* de l'application

quels paramètres. Il indique le résultat du tirage de la transition grâce à une sortie textuelle et une sortie graphique. Les graphiques générés correspondent aux figures vues en annexe A.

Outils IOSTS fait l'opération miroir et somme de comportements depuis un automate IOSTS . Il donne une sortie graphique ou un fichier IOSTS. L'opération miroir est présentée en section 6.2.2 (page 99) et l'opération somme de comportements est donnée en section 6.2.5 (page 102).

Calcul de règles fait le calcul du miroir de l'automate contrôleur pour un système donné en fonction de règles. Le programme donne une sortie graphique ou un fichier IOSTS. La sortie graphique donne une vue simplifiée ou une vue complète de l'automate contrôleur. La vue simplifiée est réalisée en appliquant l'algorithme de calcul de l'automate sans le point 9.

Grossièrement, les étapes de *Iosts Explorateur* sont d'ouvrir un fichier IOSTS, demander à l'*interpréteur de fichiers IOSTS* de créer un *Modèle* et boucler sur une demande à l'utilisateur de l'action à déclencher. À chaque boucle, une transition est déclenchée si c'est possible, puis une sortie graphique et une textuelle permettent de connaître le contexte de l'automate et les différentes actions possibles.

De manière analogue, les étapes des deux derniers programmes sont d'ouvrir

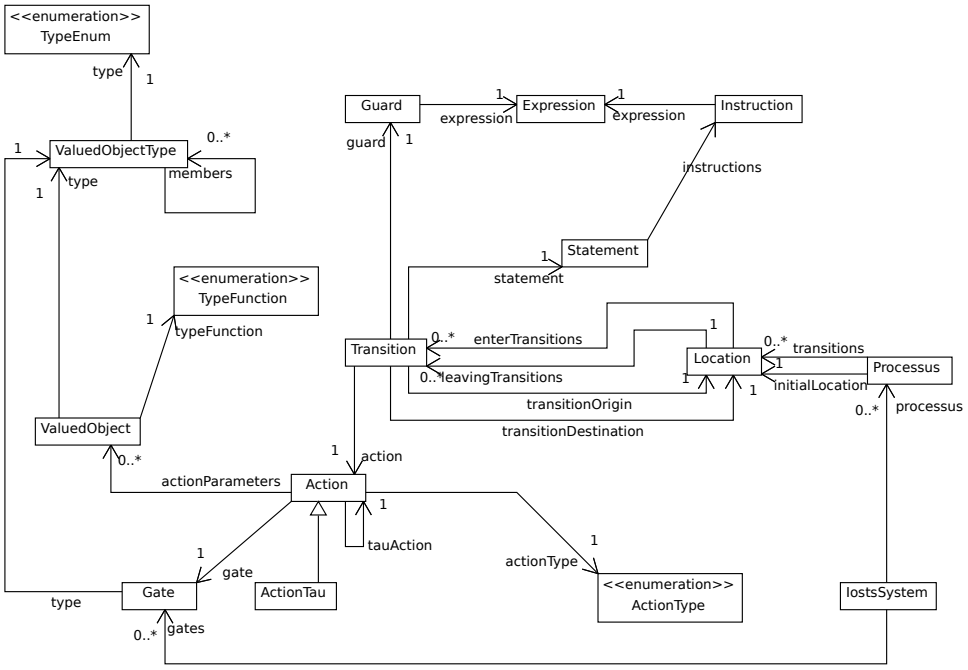


FIGURE 7.8 – Diagramme de classe du modèle de Ibyla

un fichier IOSTS, demander à l'*interpréteur de fichiers IOSTS* de créer un *Modèle*, calculer au moyen des *Outils IOSTS* un nouvel IOSTS en fonction des paramètres de la ligne de commande et sortir l'automate dans les formats souhaités.

Les autres modules sont présentés par la suite.

7.5.2 Modèle objet IOSTS

Le modèle de Ibyla correspond au modèle formel présenté dans la définition 1 page 51 avec certaines différences. La figure 7.8 présente le diagramme de classe UML du modèle. Ce modèle est adapté du modèle du programme *STG* [77], l'implantation d'un outil de validation et de test d'automates IOSTS codé en OCaml.

STG est librement accessible à l'adresse `http://www.irisa.fr/prive/ployette/stg-doc/stg-web.html`. Il est développé par F. Ployette et F.-X. Ponscarme au laboratoire IRISA/INRIA Rennes.

Les gardes (*guard*) et les *instructions* sont exprimées sous forme d'*expressions*. Les expressions sont des chaînes de caractères compatibles avec l'*Évaluateur de script*. Les *statement* sont les affectations des transitions, ce sont un ensemble d'*instructions*.

Les objets de type *IostsSystem* sont une agrégation de processus. C'est un moyen technique permettant de faire correspondre les actions partagées par différents systèmes IOSTS.

Les *actions* sont représentées par des *gate*s. Les *gates* portent le nom de l'action et la classe *action* porte la valeur de ses paramètres.

Les objets *ValuedObject* représentent la valeur d'une constante, d'une variable de processus ou d'un paramètre d'action. Ces objets sont de type *ValuedObjectType*. *ValuedObjectType* permet de définir des types simples ou complexes. Les types simples sont des chaînes de caractères, des entiers, des réels, des booléens, et des tableaux. Les types complexes sont des structures de *ValuedObjectType*.

Il faut noter que l'implantation ne prend en compte que les entiers et les booléens.

7.5.3 Modules non fonctionnels

Le module *interpréteur de fichiers IOSTS* permet de créer les objets d'un modèle IOSTS à partir d'un fichier IOSTS. Le format des fichiers IOSTS doit suivre une grammaire adaptée de celle utilisée par l'outil STG. Elle est présente en annexe C. Le programme de décomposition analytique de Ibyla est généré au moyen de l'outil *JavaCC*. Afin de générer ce programme, JavaCC a besoin de la grammaire des fichiers IOSTS et d'instructions permettant d'indiquer comment créer le système IOSTS.

La grammaire utilisée par l'implantation de Ibyla est modifiée par rapport à celle de l'outil *STG*. Ces changements sont indiqués à la fin de l'annexe C. Ces modifications sont dues à l'utilisation d'un *évaluateur de script* afin d'interpréter les *expressions* du modèle. Les expressions sont formulées dans le langage Groovy [31], un dérivé du langage Java. Nous avons changé la façon d'écrire les opérateurs booléens.

L*évaluateur de script* utilise la spécification JSR-223. Cette spécification de Java 6 permet à des scripts d'accéder à des informations statiques et dynamiques d'un programme lors de son exécution. L'API de la JSR-223 est implantée dans la bibliothèque standard de Java 6 et par la bibliothèque Apache Commons Bfs. Cette dernière bibliothèque permet d'utiliser les langages de scripts dans les programmes fonctionnant sur des JVM 1.4 et plus.

Nous avons choisi d'utiliser le langage Groovy afin de remplacer les expressions du langage des fichiers IOSTS de STG. Grâce aux spécifications JSR-223, nous pouvons à tout moment utiliser un nouveau langage de script. Par exemple, le langage ruby peut remplacer Groovy grâce à son implantation JRuby fonctionnant sur une JVM.

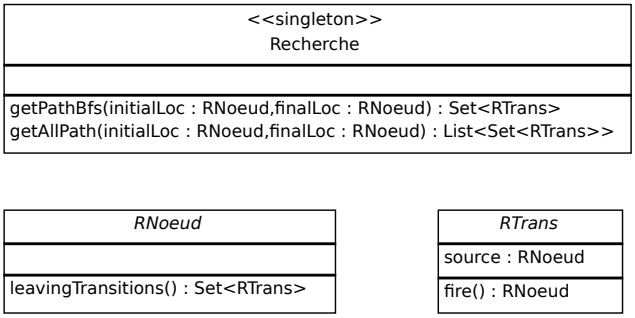


FIGURE 7.9 – Diagramme de classe du modèle de recherche et de parcours

La recherche de chemins de l'application est réalisée par l'intermédiaire d'un composant de parcours et recherche de chemin générique. C'est la rôle du module *Outils Graphes*. Comme le montre la figure 7.9 trois classes sont nécessaires pour exécuter les algorithmes. Les deux interfaces *RNoeud* et *RTrans* permettent de modéliser respectivement les localités et les nœuds de graphes. La fonction *leavingTransitions()* permet de retourner toutes les transitions ayant comme origine la localité pour laquelle la fonction est appelée. La fonction *fire* premet de déclencher une transition et de se retrouver dans la prochaine localité. Le singleton *Recherche* implante une fonction de recherche du plus court chemin et une fonction de parcours de graphe pour donner tous les chemins vers une localité.

Dans le cas de la recherche de chemin entre deux localités, la recherche est simple et les implantations de *RLoc* et *RTrans* sont évidentes. En revanche, lorsqu'il faut chercher un chemin à partir d'une localité jusqu'à un état, l'implantation est plus complexe. La fonction *leavingTransitions()* retourne un ensemble de transitions initialisées avec tous les paramètres possibles.

Nous proposons en perspectives de s'abstraire de la recherche de chemin d'une localité à un état par interprétation de garde. Cela augmenterait la vitesse de résolution des algorithmes.

Les outils disponibles dans le module *Aide IOSTS* permettent une sortie graphique et une sortie textuelle. La sortie graphique utilise la boîte à outils *graphViz* qui permet de créer des graphes en fonction d'une entrée textuelle. L'implantation de Ibyla utilise un appel système afin d'utiliser *dot*, un des outils de *graphViz*. La sortie textuelle génère un fichier de format IOSTS.

7.5.4 Module fonctionnel

La réalisation du module fonctionnel et notamment l'implantation des algorithmes de création d'automates contrôleur est codé dans le langage *Scala* [71]. Une note explicative de ce langage de programmation est disponible en annexe B (page 151). Nous avons choisi *Scala* car c'est un langage de programmation basé sur le paradigme de la programmation fonctionnelle, adapté à la mise en œuvre de systèmes mathématisés.

7.6 Synthèse

Ce chapitre décrit un exemple de chacun des trois types de règle appliquée à un ou deux dispositifs. Pour cela, il introduit un système de fenêtre oscillo-battante fictif permettant de complexifier les calculs des règles. Il introduit aussi le comportement d'un interrupteur et indique comment cet interrupteur est transformé en point de commande.

La dernière section décrit les technologies utilisées pour la réalisation d'un système logiciel expérimental mettant en œuvre les algorithmes que nous avons développés.

Conclusions et perspectives

Synthèse des résultats

Nous avons vu dans l'introduction de ce document que l'environnement ubiquitaire se développe et arrive aujourd'hui dans nos habitations. Son développement est freiné par la multitude de fournisseurs ne collaborant pas ensemble et fournissant des solutions incompatibles entre elles et non complètes. En particulier, il faut retenir que si le marché se diversifie en termes de dispositifs, il doit également se développer en terme de services.

Les projets que nous avons analysés au début de notre thèse dont les travaux portent sur la prise en compte du contexte ne s'intéressent pas à la gestion du comportement des systèmes. Ceux qui prennent en compte les capacités de commande des dispositifs domotiques s'emploient pour la plupart à modéliser des scénarios de vie dans les habitations. Ces scénarios sont en réalité une succession de commandes à envoyer lorsqu'un évènement est détecté et aucune coordination n'est assurée entre les dispositifs ni en liaison avec des besoins utilisateurs exprimés à un niveau sémantique plus élevé que la commande élémentaire d'un dispositif.

Ontologies et systèmes domotiques

La première action à entreprendre pour gérer un système domotique hétérogène est de créer un environnement virtuel où tous les dispositifs sont accessibles et où ils peuvent communiquer leurs états ou l'état de l'environnement. Il s'agit de construire une abstraction des systèmes concernés. Nous avons vu que les systèmes doivent être nécessairement abstraits lorsque l'on s'intéresse à la prise en compte

du contexte de l'installation. L'abstraction et la prise en compte du contexte sont réalisées par un outil mathématique et informatique proche de la logique du premier ordre, les ontologies.

Un environnement est composé d'éléments réels capables d'interactions comme les dispositifs et personnes, d'éléments réels statiques comme les lieux de vie, les meubles, et d'éléments virtuels capables d'interaction comme des services. Chacun des éléments capables d'interactions dispose de plusieurs états, ainsi que de moyens de communication. Les technologies de communication de ces éléments et l'accès à leurs fonctionnalités sont hétérogènes.

Les ontologies sont un moyen permettant de stocker la connaissance. En ce sens, les fonctionnalités et les états de tous les éléments capables d'interaction peuvent être stockés au sein d'une ontologie. Il en est de même de la connaissance des lieux de vie ou de la localisation des meubles.

Les ontologies ont un fondement basé sur la logique du premier ordre. Il est ainsi possible d'interpréter algorithmiquement cette connaissance et ainsi se rapprocher de la pensée humaine.

Des nombreux travaux qui existent sur les systèmes domotiques, nous avons retenu l'approche DogOnt qui met en jeu une ontologie explicite du système domotique.

Nos travaux (chapitre 4), menés au sein de l'entreprise Overkiz ont consisté à proposer une solution ontologique pour la gestion des dispositifs. Nous avons participé à l'analyse des différents écosystèmes (Open Source ou propriétaires) afin de les insérer dans le système d'information et de contrôle de Overkiz.

Nous avons étudié deux écosystèmes d'objets communiquant, UPnP et ZigBee, et extrait les bonnes pratiques de ces modèles. Nous avons pris en compte l'aspect générique du modèle UPnP qui permet d'utiliser des fonctionnalités sous forme de service et le modèle de présentation des fonctionnalités des dispositifs Zigbee. Les objets communicants sont ainsi modélisés par un ensemble de fonctionnalités et d'états.

En nous appuyant sur DogOnt, et en appliquant la méthodologie fondée sur les ontologies, nous avons collaboré au déploiement d'une version adaptée de DogOnt au sein du système d'information de Overkiz. En effet, même si les ontologies sont une approche étudiée depuis de nombreuses années et qui dispose de nombreux outils, la création d'une ontologie et la vérification de sa cohérence restent un processus manuel ou semi-manuel très sujet à erreurs.

Comportement

Si les ontologies constituent un outil adapté à l'abstraction des systèmes de domotique, elles sont pour l'instant essentiellement cantonnées à la description de ces systèmes. Elles ne permettent pas de gérer, i.e. de d'observer et de commander,

les interactions donc les liens qui existent entre les états, les commandes et les notifications.

Nous nous sommes attachés dans cette thèse à proposer des outils mathématiques permettant de modéliser le comportement d'un environnement composé d'actionneurs, de capteurs et d'interfaces utilisateurs. Le comportement est formalisé (chapitre 5) au moyen de la théorie des automates étendus et plus particulièrement la théorie des systèmes de transitions symboliques à entrées-sorties (*Input Ouput Symbolic Transition System*, IOSTS). Nous avons étudié plus particulièrement les environnements automatisés d'habitations. C'est en effet la modélisation du comportement qui permet de connaître les liens qui existent entre les états d'un dispositif, ses commandes et ses notifications et donc de contrôler le système domotique.

Nous avons introduit dans nos modèles IOSTS, la prise en compte de l'indéterminisme qui résulte du fonctionnement du système, lorsque des actions non déclenchées par le contrôle sont observées. Nous utilisons pour cela des modèles de comportements sur détection d'évènements, et nous prenons en compte la notion d'état interdit.

Nous avons également introduit au niveau IOSTS, un modèle des interactions entre dispositifs, en particulier pour rendre compte des relations dispositif de commande (un bouton par exemple) - dispositif commandé (une lampe par exemple). Ce modèle est également capable de restituer les interactions système-utilisateur, en particulier dans les souhaits de fonctionnement et les contraintes que ce dernier peut exprimer.

À partir de ces modèles, nous avons élaboré (chapitre 6) un algorithme de synthèse d'un contrôleur capable de piloter de manière cohérente, et en tenant compte des besoins exprimés par l'utilisateur, un système complexe de domotique. L'algorithme est en fait un ensemble de trois algorithmes correspondant à trois situations. En particulier, et cela requiert une suite significative d'étapes, notre construction permet d'obtenir un contrôleur guidant le système vers un ou des états souhaités (un but) à partir d'un autre état. L'opération de synthèse est réalisée grâce à un automate de comportement du système et de règles. La conception de ces algorithmes a nécessité la définition d'opérateurs sur IOSTS permettant de manipuler les automates des systèmes. À l'inverse des automates de comportement, le contrôleur indique quelles sont les commandes à envoyer. Ainsi, lorsque le système est dans un état où le contrôleur peut émettre une commande, le système contrôleur se doit d'envoyer cette commande.

Enfin, nous avons validé la synthèse de contrôleurs et les opérateurs associés, par l'intermédiaire d'une implantation -preuve de concept- fonctionnant sous une machine virtuelle Java.

Perspectives

L'utilisation des règles afin de créer un comportement montre la facilité d'utilisation du modèle proposé dans cette thèse. Cependant, certaines améliorations ou extensions sont possibles.

Aspects scientifiques et techniques

Deux directions nous semblent pertinentes en prolongement des travaux que nous avons menés.

Prise en compte du temps physique

Notre modèle ne prend pas en compte le temps physique, i.e. le temps perçu par l'utilisateur et les dispositifs. Ainsi, il n'est pas possible d'exprimer une commande telle que : « à 20h, obturer la pièce de vie », alors que l'on peut gérer la commande « obturer la pièce de vie ». Intégrer la notion de temps permettrait d'étendre les mécanismes de règles à des scénarios de vie, par exemple permettre le déclenchement d'actions périodiques. L'introduction du temps peut se concevoir d'au moins deux manières. Soit recourir au modèle des automates temporisés [3] ; ceci modifie profondément la sémantique sous-jacente, et risque de conduire à des problèmes indécidables. Soit introduire un « dispositif virtuel » qui emmétrait des notifications à intervalles temporels réguliers, représentant ainsi une horloge du système contrôlé. Pour autant, une étude plus approfondie du mécanisme reste à mener pour s'assurer de la faisabilité de cette solution.

Contrôle de l'explosion combinatoire

Il est clair que la synthèse du comportement d'une installation est consommatrice en mémoire et en temps de calcul. Dès que l'on introduit n dispositifs, nous obtenons un espace potentiel de localités qui est le produit cartésien des n espaces de localités de ces dispositifs. La situation est encore aggravée par le fait qu'on doit déterminer des chemins dans un graphe dont les noms sont ces localités. Même si le problème général devient très complexe lorsque n augmente, plusieurs stratégies, au niveau du modèle peuvent être envisagées pour contenir cette explosion combinatoire.

Interprétation des gardes

L'algorithme de recherche de chemin pour atteindre un état précis *trouverCheminJusquAEtat* est basé sur l'exploration de l'ensemble de l'espace des paramètres des actions des transitions. Il est utilisé par le calcul des contrôleurs ECB et par l'étape 7 du calcul des contrôleurs de contraintes présent à la section 6.3.2.

Dans la section 6.3.2, nous avons posé l'hypothèse 6.1 qu'une transition amenant à une localité peut balayer toutes les valeurs du domaine de toutes les variables. Cette hypothèse permet de simplifier la fonction *sélectionnerChemins* qui permet de faire le choix des chemins nécessaires pour aller d'une localité à une autre localité. Le choix des chemins est utilisé par l'étape 6 du calcul des contrôleurs de contrainte.

Ces deux problèmes peuvent être résolus grâce à la gestion des domaines des variables pour les localités et l'interprétation des gardes et des affectations régissant les transitions. La fonction $trouverChemins : (L_{\mathcal{S}}, L_{\mathcal{S}}) \rightarrow ListeChemins$ est alors modifié afin de retourner une liste de chemins amenant à une localité, chaque chemin est donné avec le domaine des variables qu'il permet de commander.

Dans le premier cas, la fonction $trouverCheminJusquAEtat$ n'est alors plus utilisée, elle est remplacée par la fonction $trouverChemins(l^o, l^d) = c$ puis $sélectionnerCheminsSecours$. Cette dernière fonction choisit le premier chemin le plus court permettant d'aller à l'état destination. Le chemin choisi est adapté pour donner les bonnes valeurs aux variables.

Dans le deuxième cas, la sélection des chemins se fait en maximisant le domaine destination et en minimisant la taille des chemins. Des gardes sont affectées aux premières transitions des chemins lorsqu'ils sont copiés dans le système.

Interprétation des affectations

Le modèle proposé en définition 1 ne permet pas des affectations calculées. En effet, la fonction d'affectation $A : Dom(V) \mapsto Dom(V)$ d'une transition est un ensemble d'expression de la forme $(x := A^x)_{x \in V}$ tel que pour tout $x \in V$, $A^x \in V \cup sig(a)$. C'est-à-dire qu'une variable peut-être affectée à la valeur d'une variable ou à la valeur d'un paramètre.

Cette limitation permet de faciliter la mise en place du point précédent. Nous pourrions calculer les domaines de transition ou les affectations sont plus souples grâce à la technique d'interprétation abstraite. Elle permet de connaître les domaines destinations des transitions même lorsque l'affectation est un calcul. En revanche, une telle méthode engendre des simplifications lorsque le calcul est compliqué. Il faudrait limiter le pouvoir expressif des calculs des affectations afin de ne jamais obtenir de simplifications.

Optimisation des algorithmes de graphe

En liaison avec une implantation plus efficace de nos algorithmes, il est probable que l'emploi d'heuristiques adaptées pour la recherche des chemins dans les IOSTS réduira significativement le temps d'exécution ce ceux-ci.

D'un point de vue industriel

Du prototype à la mise en production
Le prototype développé ne peut pas être déployé en l'état pour une application en entreprise. Au-delà des problèmes d'interface utilisateur, il est nécessaire de contenir les effets de l'explosion combinatoire indiqués ci-dessus. De plus, une industrialisation du code devra être menée. L'intérêt du développement d'une version industrielle de notre solution est particulièrement net pour la prise en compte de *scenarii* de fonctionnement de la maison plus élaborés (règles de comportement) que ceux actuellement disponibles.

Services de haut niveau sémantique
L'une des applications des ontologies que nous n'avons pas abordé est de permettre l'élaboration de services à fort pouvoir d'expression. Dans le domaine du WEB, on parle du « WEB sémantique ». Il s'agit d'augmenter les descriptions, essentiellement syntaxiques, actuellement disponibles pour les services WEB, avec des termes se rapportant au sens du service et non plu seulement à sa syntaxe d'appel. Les services utilisant cette technologie sont fondés sur les ontologies. Ils sont plus proches de la compréhension d'un utilisateur et peuvent être manipulés par des éléments calculatoires. Des extensions du pouvoir d'expression de nos modèles, par exemple en termes de contraintes ou de définition des états du système, pourraient être fondées sur le même type de méthodes, puisque nous employons déjà des ontologies pour décrire les dispositifs.

Annexes

Annexe

A

Automates de comportement calculés

Cette annexe donne un exemple de sorties graphiques de quelques automates contrôleur. La représentation de ces automates n'est pas la même que celle présente dans le reste du mémoire. Ces images sont générées par l'outil *dot* de l'application *graphViz*.

La première différence est la représentation des transitions. La localité d'origine est représentée par la source de la flèche entrante. De même, la localité de destination est symbolisée par la destination de la flèche sortante. Les actions, gardes et affectations des transitions sont représentées textuellement dans un carré. La première ligne est la garde de la transition, elle est précédée du mot *if*. La deuxième ligne est l'action, elle est précédée du mot sync. Enfin, la dernière ligne est l'ensemble des affectations, elles sont représentées à l'intérieur d'un d'accolade précédée du mot *do*.

La deuxième différence est que toutes les localités sont représentées par un ovale. Dans cette représentation, il n'y a pas de différence symbolique entre les localités stables, virtuelles et transitoires. Seul le nom des transitions donne un indice sur la nature de la localité.

A.1 Fenêtre oscillo-battante

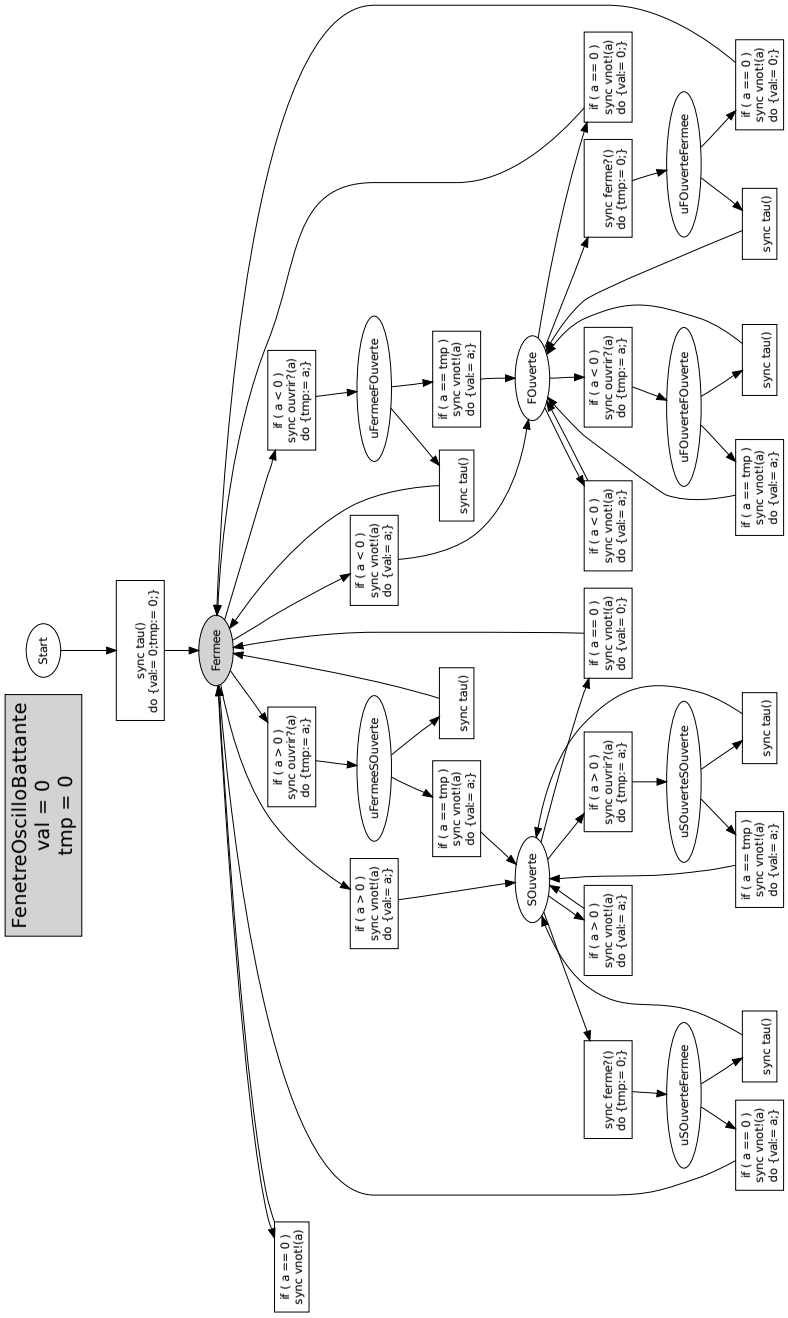


FIGURE A.1 – Automate de comportement d'une fenêtre oscillo-battante

A.2 Miroir d'un contrôleur d'ECA

La règle ECA de la figure A.2 est : btNot() ; $val > 0 \vee val < 50$, ouvrir(100).

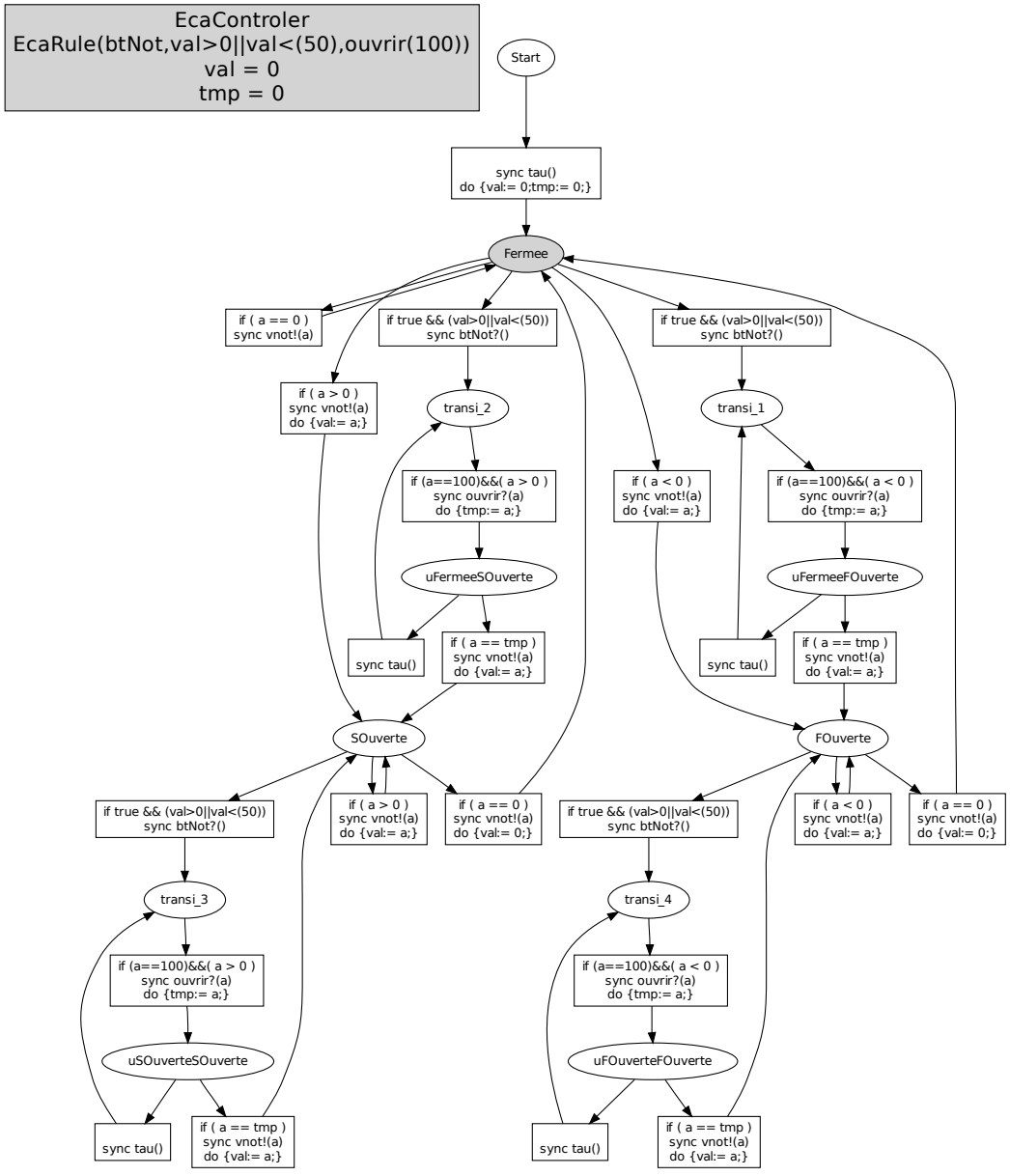


FIGURE A.2 – Miroir de l'automate de comportement du contrôleur

A.3 Miroir d'un contrôleur d'ECB

La règle ECB de la figure A.3 est : btNot() ; $val > 50 \vee val < (-50)$; FOuverte ; $val = -50$.

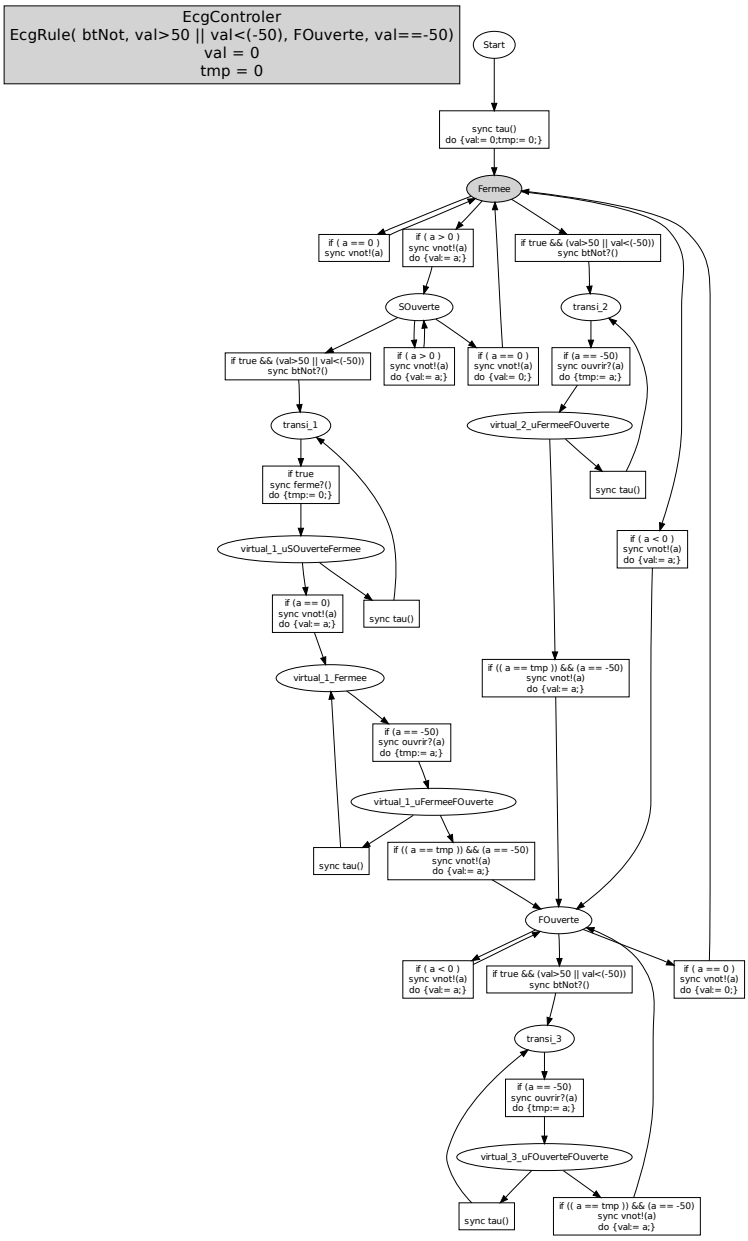


FIGURE A.3 – Miroir de l'automate de comportement du contrôleur

A.4 Miroir d'un contrôleur de contrainte

La règle de contrainte de la figure A.5 est : $\langle val > 0 \wedge val < (-10), SOuverte, val = 65 \rangle$.

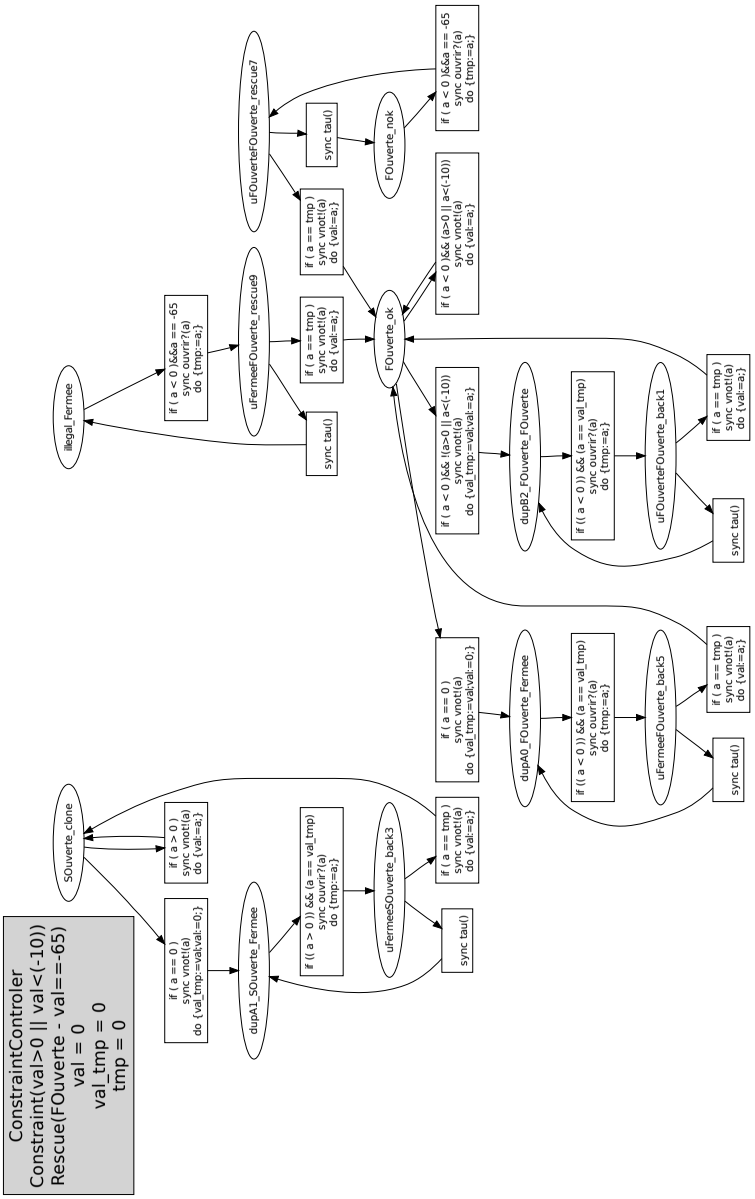


FIGURE A.4 – Vue simplifiée du miroir de l'automate de comportement du contrôleur de contrainte simplifié

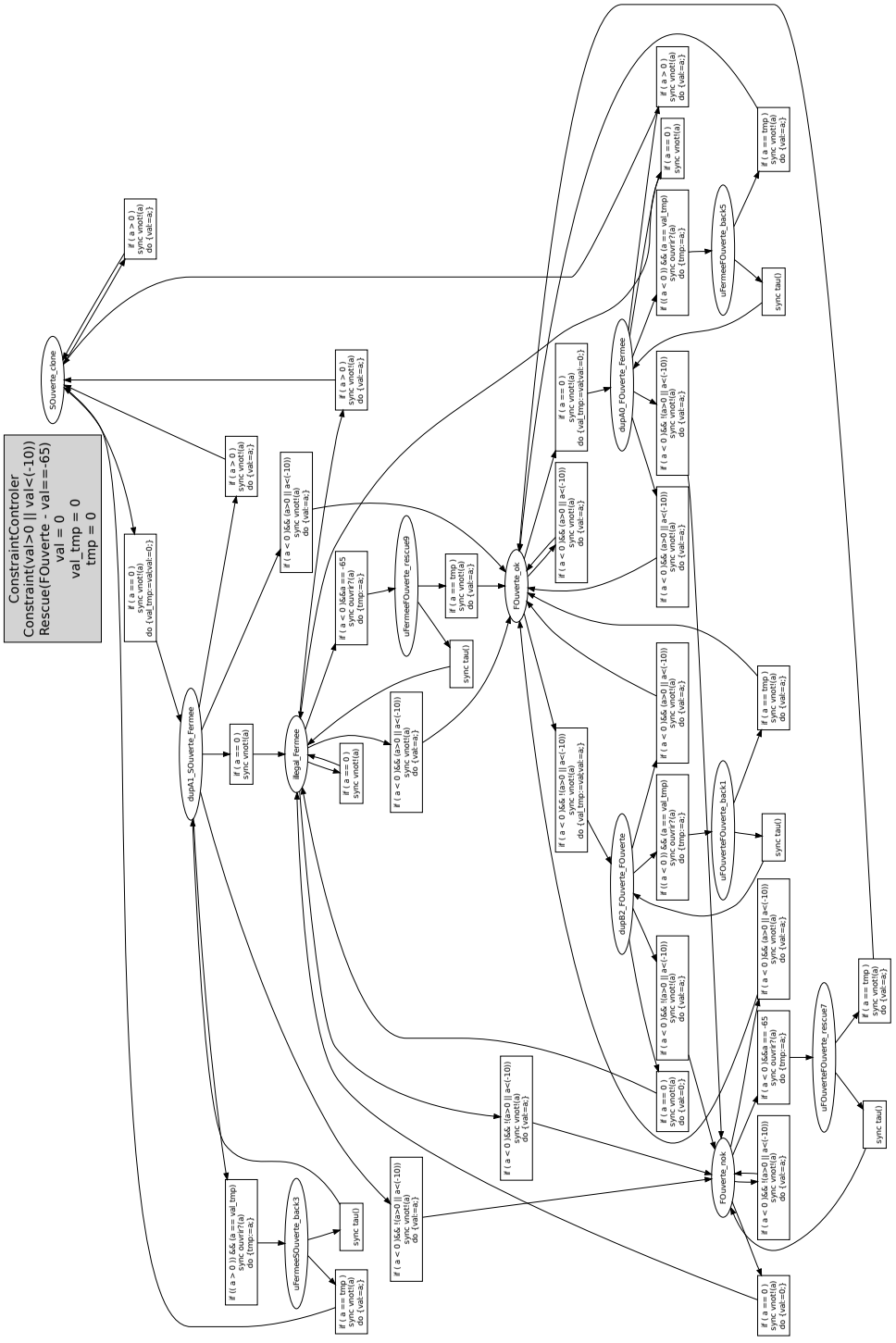


FIGURE A.5 – Miroir de l'automate de comportement du contrôleur de contrainte

Annexe

B

Présentation de scala

Scala est un langage de programmation intégrant le paradigme fonctionnel et orienté objet. Ces deux paradigmes sont complémentaires, le fonctionnel permet de décrire l'aspect algorithmique d'un programme et l'orienté objet plus adapté pour décrire les différentes structures des données.

Scala peut être compilé pour deux types de machine virtuelle : celle de Java (JVM) et celle de Microsoft (CLI). L'utilisateur souhaitant exécuter son application uniquement sur la JVM pourra utiliser les bibliothèques écrites en Java.

B.1 Aspect orienté objet

Le principe de la programmation orientée objet (POO) est de fournir un conteneur qui associe une structure de donnée avec les opérations réalisables sur ces données et un ensemble de principes permettant de réutiliser du code. La définition de ce conteneur est donnée dans des éléments que l'on nomme classe. L'instance d'une classe correspond à un objet, c'est un élément qui contient des données. L'objet est utilisé par un programme lors de l'exécution.

La POO permet de décrire les structures de données d'un programme au moyen de l'héritage. L'héritage est le principal principe qui permet de réutiliser du code. Il consiste à reprendre les données et les opérations d'une classe "parente" pour les donner à une autre classe que nous nommerons "enfant". L'avantage est que la classe enfant est vue comme une classe disposant des mêmes comportements que

ses parents. L'héritage de plusieurs parents pose de nombreux problèmes, mais le principe de "voir" une classe disposant de caractéristiques communes avec d'autres classes permet de simplifier la création de programme. Pour régler le problème de l'héritage multiple, Java introduit le principe d'interfaces. Les interfaces sont une sorte de contrat signé par une classe et qui permet d'appeler des comportements spécifiés par d'autres classes.

Scala pousse le principe des interfaces plus loin en permettant de donner l'implémentation des comportements (en plus de la spécification) par l'intermédiaire des *traits*. Les traits différents de l'héritage multiple par le choix fait pour régler les problèmes inhérents à l'héritage multiple. De plus, les traits peuvent être mixés avec des objets, c'est-à-dire qu'une classe n'est pas obligée d'hériter d'un trait pour qu'un objet puisse recevoir les caractéristiques de ce trait

Un second principe de réutilisation de code est la paramétrisation des types des classes, la généricité des classes. Il est introduit dans la version 1.5 de Java et Scala reprend les grands principes des classes génériques de Java. Une classe ou un trait générique définit ou spécifie un comportement sur des données dont au moins une possède un type n'étant pas connu d'avance. Ce principe pose des problèmes lorsque l'on souhaite connaitre si deux objets d'une classe générique héritent l'un de l'autre ou non. Le domaine de l'informatique décrit ce problème comme la variance des types. Java introduit cette notion pour les classes génériques en prenant par défaut le choix de covariance pour ces classes. Ce choix est fait pour que les nouvelles JVM soient compatibles avec les codes d'anciens programmes. Quant à Scala, il fait le choix de prendre les classes/traits génériques comme invariants.

Il existe trois formes : la covariance, la contravariance et l'invariance :

- Lorsqu'une classe générique est covariante, une instance typée d'un type A est un objet enfant d'une instance typée d'un type B où B est enfant de A.
- Lorsqu'une classe générique est contravariante, une instance typée d'un type A est parent d'une instance typée d'un type B où B est enfant de A.
- Lorsqu'une classe générique est invariante, une instance typée d'un type A n'a aucun rapport avec une instance typée d'un type B où les types A et B ne sont pas identique.

Un autre aspect important de Scala est la possibilité de définition d'objets compagnons. Ces objets compagnons cadrent la création et l'utilisation des objets singletons.

B.2 Aspect fonctionnel

Le paradigme de programmation fonctionnel (FP) est plus ancien que la POO, pourtant il est moins utilisé que ce dernier. Cette différence d'utilisation peut être

expliquée par la facilité de compréhension de la POO ainsi que les performances des programmes impératifs. Mais cette tendance est peut-être en train de changer. En effet, FP est plus expressive et permet d'exprimer la structure de résolutions d'un problème de manière plus concise.

FP est construite sur les principes de *first-class function* et de la possibilité de faire des *closures*. First-class function signifie que les fonctions sont traitées comme n'importe quelle donnée de variables. Plus formellement, en FP les fonctions sont traitées comme des éléments de premier ordre. À cet effet, une fonction peut être déclarée dans le corps d'une autre fonction et peut être donnée en retour d'un appel d'une autre fonction. Ce premier principe est complété par les closures. Ce dernier est un mécanisme qui capture les références des symboles utilisées dans une fonction qui ne sont ni des variables ni des paramètres de cette fonction.

La FP peut être représentée comme un flux d'opérations sur des données, plutôt qu'une succession de modifications de données. Elle considère l'évaluation de ses fonctions comme l'application de fonctions mathématiques qui, à partir de données d'entrée, génère une nouvelle donnée de sortie. Les données non modifiables sont dites immuables. L'évaluation d'une fonction mathématique donne toujours le même résultat lorsque l'on donne les mêmes données d'entrée, ce principe est la propriété de transparence référentielle.

FP encourage l'utilisation des données immuables ainsi que la transparence référentielle et les fonctions récursives. Scala propose aussi des structures de données et de contrôles itératives lorsque le besoin s'en fait sentir (optimisation, algorithme itératif plus simple, ...). Par exemple, le type immuable ListSet[T] de la bibliothèque Scala utilise un type mutable (ListBuffer[T]) pour des raisons de performances.

Pour terminer, Scala définit seulement les contrôles suivants : *if*, *for*, *while*, *try* et *match*. Chacune de ces structures renvoie une donnée (*while* renvoie toujours Unit). *If* et *try* renvoient la dernière valeur utilisée. *Match* permet d'énumérer les possibilités d'une variable (ou d'un N-Uple de variables) et donner les actions à entreprendre pour chacun des cas. Cet élément reverra la valeur de retour des actions. *Match* correspond au switch de java mais en plus général. En effet, il peut tester l'égalité de valeurs, de types, de valeur de paramètre d'un objet ...Cet aspect de Scala est limité par JVM 1.6 car cette dernière ne dispose pas de l'information de type lors de l'exécution d'un programme (problème d'*erasures*, en partie réglé avec les objets Manifest[T]). Et enfin l'élément *for* qui est une aide syntaxique et permet de décrire une succession de manipulation. Pour cela, *for* utilise :

- des générateurs (*map*, *flatMap*, *forEach*),
- des définitions (permet de définir des variables)
- des filtres (*withFilter*)

B.3 Synthèse

Pendant longtemps, FP fut réservé au monde de la recherche et de l'enseignement. Aujourd'hui, de nouveaux langages de programmation fonctionnelle arrivent sur le marché à destination des logiciels d'entreprise. Pour là plupart hybrides, ils permettent d'utiliser des objets mutables et des structures de contrôle itératives.

Cette annexe présente les deux principaux paradigmes de programmation utilisés dans Scala. La POO permet de structurer les données alors que la FP permet de décrire le contrôle des données de manière plus mathématique. Certains points font un lien avec les choix de Scala vis-à-vis de Java comme l'erasure ou les classes génériques.

Annexe

C

Grammaire des fichiers IOSTS

Cette annexe présente la grammaire des fichiers IOSTS.

Déclaration d'un système

```
<system> ::= system <system-id> ;
             [const <constant>+]
             [type <type>+]
             gate <gate>+
             <process>+
```

Constante des système

```
<constant> ::= <constant-id> = <constant-body> ;
<constant-body> ::=  true
                  |  false
                  |  <integer>
                  |  <constant-id>
```

Types disponible pour un système.

```
<type> ::= <type-id> = <type-body> ;
<type-body> ::= enum <enum-decl-list>
```

```
    | range [-]{<integer>|<constant-id>} .. [-]{<integer>|<constant-id>}
    | record { <field-id> : <type-id> }+ end
    | array [ [<integer>  | <ident>] ]of <type-id>
```

Gate du système.

```
<gate> ::= <gate-id> [ { ( <type-id> { <type-id> }*) }0/1 ] ;
```

Déclaration des processus.

```
<process> := process <process-id> ;
             input <gate-id>+
             output <gate-id>+
             internal <gate-id>+ [tau]
             [parameters <params>+]
             [variables <vars>+]
             state <state>+
             transition <transition>+
```

Paramètres, variables et états d'un processus.

```
<params>   ::= <ident-name> : <type> ;
<vars>     ::= <ident-name> : <type> ;
<state>    ::= init : <state-id> ;
             | <state-id>
<state-id> ::= <ident>
             | <integer>
```

Transitions d'un processus.

```
<transition> ::= from <state-id>
                 [ <guard> ]
                 [ <action> ]
                 [ <statement> ]
                 to <state-id> ;
```

Garde, action et type d'action d'un processus.

```
<guard>           ::= if <expression>

<action>          ::= sync <action-type>
<action-type>     ::=  <internal-action>
                    | <other-action>
<internal-action> ::= <gate>
                    | tau
<other-action>    ::= <gate> { ?  | ! } <io-list>
<io-list>         ::= ( { <mess-id> { , <mess-id>}+ }* )
<mess-id>         ::= <ident>
                    | <constant-id>
```

Affectation des transitions.

```
<statement> ::= do { <inst> {| <inst>}+ }
              | do <inst>
<inst>      ::= <expression> := <expression>
```

Expression d'une affectation ou d'une garde.

```
<expression> ::= <constant-value>
         | <name>
         | + <expression>  | - <expression>
         | not <expression>
         | <expression> [ <expression> ]
         | <expression> . <expression>
         | <expression> -> <expression>
         | ( <expression> )
         | <expression> <binop> <expression>
<name> ::= IDENT
```

Binop de la version STG.

```
<binop> ::= or  | and  | = | <>
   | <  | <=  | >=  | >
   | +  | -  | * | /  | %
```

Binop de la version de Ibyla.

```
<binop> ::= '||' | &&
   | =  | == | <> | !=
   | <  | <= | >= | >
   | +  | -  | *  | /  | %
```

Glossaire

Glossaire

A

Acteur

Un acteur est une entité utilisant un système. Il peut être un autre système ou un utilisateur. Un acteur est capable de réaliser des actions spécifiques dans un environnement donné. Il est capable de communiquer directement avec d'autres agents de l'environnement. 26, 27, 34, 35, 49

API

Les interfaces de programmation ou API (*Application Programming Interface*) sont utilisées par des programmes pour accéder à des fonctionnalités délivrées par un autre programme. En général, les API exposent un ensemble de données et de fonction pour faciliter les interactions entre programmes et leur permettent d'échanger de l'information. 17, 22

C

Composant

Un composant dans le domaine du logiciel est un élément capable de communiquer avec d'autres composants. 13

E

Ecosystème

Un écosystème est un ensemble de dispositifs communicants entre eux avec des règles communes. Ce terme se substitue à infrastructure réseau. 10, 15, 21

Evènement Condition Changement (ECC)
Une règle *Évènement Condition Changement* exprime une action à entreprendre lorsqu'un événement apparait et que le système est dans un certain état. 92, 105

F

Fonctionnalités primaires
Les fonctionnalités primaires d'un objet sont toutes les commandes qui permettent de faire fonctionner un dispositif et les notifications qui permettent de connaître l'état d'un dispositif. En d'autres termes, les fonctionnalités primaires permettent de changer l'état d'un dispositif et de connaître son état. Par exemple, c'est la possibilité d'ouvrir ou fermer un volet. 74, 86

Fonctionnalités secondaires
Les fonctionnalités secondaires d'un dispositif sont des propriétés des dispositifs qui sont sous-entendues ; elles ont une signification pour un utilisateur. Par exemple, c'est la propriété d'isolation thermique d'un volet fermé. 74, 86

Framework
Un framework est un cadre de développement logiciel fournissant des bibliothèques. Il s'occupe d'une ou plusieurs préoccupations. 11, 14, 17, 23

I

Intergiciel
Un intergiciel, en anglais middleware, est un système logiciel visant à rendre transparent la répartition des différents composants d'une application. 4, 11, 12, 19, 23

K

KNX
Konnex ou KNX est un protocole de communication datant de 2002. KNX est une amélioration du bus EIB (European Installation Bus) créé en 1987 par différents constructeurs européens. En 2006, le protocole et ses médias de communication (paires torsadées, courant porteur, radio, et IP) ont été définis par la norme internationale ISO 14543-3. Son site internet est : `http://www.knx.org`. 9

M

Micrologiciel
Un micrologiciel est un programme spécifique embarqué dans un système. Ce programme gère à la fois les entrées/sorties et les algorithmes permettant de produire la fonctionnalité système. 16

O

OSGi
Plate-forme logicielle permettant de gérer dynamiquement le cycle de vie de composants logiciels s'exécutant sur une machine virtuelle java. 15, 19, 21

S

SOAP
Protocole d'échange de messages pour l'utilisation de services web. Les messages sont construits avec un format XML les rendant utilisables par tous les environnements. 19

U

UPnP
UPnP est un ensemble de protocoles de communication basés sur IP ayant pour objectif de mettre en relations différents périphériques. Son site internet est : http://www.upnp.org/. 14, 15, 17–19

W

WS
Les services web sont une méthode de communication permettant l'interaction entre machines. Ils mettent en oeuvre un serveur hébergeant des fonctionnalités et des clients exploitants ces fonctionnalités. Un fichier particulier permet de décrire des fonctionnalités d'un domaine et leur localisations. Le site internet de la spécification est : http://www.w3.org/2002/ws/. 18

Z

Zigbee
Zigbee est un protocole de communication radio utilisé pour échanger des données grâce à un média de communication sans fil. Les premières spécifications datent de 2004. Elles prennent en compte la faible puissance électrique et calculatoire des dispositifs et permettent de sécuriser les transmissions de données. Son site internet est : http://www.zigbee.org. 14, 15, 17, 86

Bibliographie

[1] ZigBee Alliance. Zigbee home automation public application profile v1.0, 2007.

[2] ZigBee Alliance. Zigbee home cluster library specification, 2007.

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[4] Colin Atkinson, Matthias Gutheil, and Kilian Kiko. On the relationship of ontologies and models. In *WoMM*, pages 47–60, 2006.

[5] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics as ontology languages for the semantic web. In *Festschrift in honor of Jörg Siekmann, Lecture Notes in Artificial Intelligence*, pages 228–248. Springer-Verlag, 2003.

[6] John Bateman, Alexander Castro, Immanuel Normann, Omar Pera, Leyla Garcia, and Jose-Maria Villaveces. OASIS - Rapport D1.2.1 Common hyper-Ontological Framework, Jan 2009.

[7] Françoise Baude, André Bottaro, Jean-Michel Brun, Antonin Chazalet, Arnaud Constancin, Didier Donsez, Levent Gürgen, Philippe Lalanda, Virginie Legrand, Vincent Lestideau, Sylvain Marié, Cristina Marin, Alain Moreau, and Vincent Olive. Extension de passerelles OSGi pour les domaines de la distribution. *OSGi Workshop in conjonction with UBIMOB 06, 3rd French-speaking*, September 2006.

[8] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. Technical report, W3C, http ://www.w3.org/TR/owl-ref/, February 2004.

[9] Dario Bonino, Emiliano Castellina, and Fulvio Corno. Dog : An ontology-powered osgi domotic gateway. In *ICTAI*, pages 157–160. IEEE Computer Society, 2008.

[10] Dario Bonino and Fulvio Corno. Dogont - ontology modeling for intelligent domotic environments. In *International Semantic Web Conference*, pages 790–803, 2008.

[11] Dario Bonino and Fulvio Corno. Rule-based intelligence for domotic environments. *Automation in Construction*, In Press, Corrected Proof, 2009.

[12] Jorge Cardoso. The semantic web vision : Where are we ? *IEEE Intelligent Systems*, 22(5) :84–88, 2007.

[13] Philipp Cimiano and Johanna Völker. Text2onto - a framework for ontology learning and data-driven change discovery. In Elisabeth Metais Andres Montoyo, Rafael Munoz, editor, *Proceedings of the 10th International Conference on Applications of Natural Language to Information Systems (NLDB)*, volume 3513 of *Lecture Notes in Computer Science*, pages 227–238, Alicante, Spain, Juni 2005. Springer.

[14] Commissariat général au développement durable. L'énergie en France - Chiffres clés. Technical report, Ministère de l'Écologie, de l'Énegie du Développement durable et de l'Aménagement du territoire, 2008.

[15] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8) :558–574, August 2007.

[16] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Validation of reactive systems. In S. Merz and N. Navet, editors, *Modeling and Verification of Real-TIME Systems - Formalisms and software Tools*, chapter 2, pages 51–76. Hermès Science, January 2008.

[17] Camille Constant. *Génération automatique de test pour modèles avec variables ou récursivité*. PhD thesis, IFSIC, Université de Rennes 1, 2008.

[18] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5 :4–7, 2001.

[19] Dog. Domotic OSGi Gateway, November 2011. http://elite.polito.it/.

[20] Didier Donsez. On-demand component deployment in the upnp device architecture. In *Proc. of 4th IEEE Consumer Commuinications and Networking Conference (CCNC) 2007, Las Vegas,11-13 Janvier 2007*, 2007.

[21] Jérémy Dubreil. Non-interference on symbolic transition system. Master's thesis, Uppsala University, February 2006.

[22] eyeOS v2.0. The open source cloud's web desktop, February 2010. http://fr.eyeos.org/.

[23] Álvaro Fides-Valero, Matteo Freddi, Francesco Furfari, and Mohammad-Reza Tazari. The persona framework for supporting context-awareness in open distributed systems. In *Proceedings of the European Conference on Ambient Intelligence*, AmI '08, pages 91–108, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Blaz Fortuna, Marko Grobelnik, and Dunja Mladenić. Semi-automatic data-driven ontology construction system. *Proceedings of the 9th international multi-conference information society*, pages 223–226, 2006.

[25] Epameinondas Gasparis, Jonathan Nicholson, and Amnon H. Eden. Lepus3 : An object-oriented design description language. In *Diagrams*, volume 5223 of *Lecture Notes in Computer Science*, pages 364–367. Springer, 2008.

[26] Michael Genesereth, Richard E. Fikes, Ronald Brachman, Thomas Gruber, Patrick Hayes, Reed Letsinger, Vladimir Lifschitz, Robert Macgregor, John Mccarthy, Peter Norvig, and Ramesh Patil. Knowledge interchange format version 3.0 reference manual, 1992. http ://www-ksl.stanford.edu/knowledge-sharing/papers/kif.ps.

[27] John H. Gennari, Mark A. Musen, Ray W. Fergerson, William E. Grosso, Monica Crubézy, Henrik Eriksson, Natalya F. Noy, and Samson W. Tu. The evolution of protégé an environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud.*, 58 :89–123, January 2003.

[28] Nikolaos Georgantas, Valerie Issarny, Sonia Ben Mokhtar, Sebastien Bianco, Graham Thomson, and Pierre-Guillaume Raverdy. Middleware architecture for ambient intelligence in the networked home. *Handbook of Ambient Intelligence and Smart Environments*, pages 1139–1169, 2010.

[29] Anne Gérodolle and André Bottara. Osgi et le projet IST Amigo. *OSGI Workshop in conjonction with UBIMOB 06, 3rd French-speaking*, 2006.

[30] Paul Grace, Gordon S. Blair, and Sam Samuel. ReMMoC : A reflective middleware to support mobile client interoperability. In *Proc. International Symposium of Distributed Objects and Applications (DOA'03)*, 2003.

[31] Groovy. http://groovy.codehaus.org/, November 2011.

[32] Object Management Group. Ontology definition metamodel (omg) version 1.0. Technical Report formal/2009-05-01, Object Management Group, 5 2009.

[33] Thomas R. Gruber. Ontolingua : A mechanism to support portable ontologies. Technical report, Knowledge Systems, AI Laboratory, 1992.

[34] Tom R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, August 1993. Kluwer Academic Publishers.

[35] Tom R. Gruber. *Ontology*, chapter Ontology, pages 1963–1965. Springer-Verlag, 2009.

[36] Nicola Guarino. Formal ontology and information systems. In *International conference on formal ontology in information systems FOIS'98*, pages 3–15, trento, italie, June 1998.

[37] Nicola Guarino and Pierdaniele Giaretta. Ontologies and Knowledge Bases : Towards a Terminological Clarification. *Towards Very Large Knowledge Bases : Knowledge Building and Knowledge Sharing*, pages 25–32, 1995.

[38] Nicola Guarino and Christopher A. Welty. An overview of ontoclean. In Steffen Staab and Dr. Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 201–220. Springer Berlin Heidelberg, 2009.

[39] Dominique Guinard, Mathias Fischer, and Vlad Trifa. Sharing using social networks in a composable web of things. In *Proceedings of the 1st IEEE International Workshop on the Web of Things (WoT 2010) at IEEE PerCom 2010*, Mannheim, Germany, March 2010.

[40] Dominique Guinard and Vlad Trifa. Towards the web of things : Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, April 2009.

[41] Levent Gurgen, Claudia Roncancio, Cyril Labbé, André Bottaro, and Vincent Olive. SStreaMWare : a service oriented middleware for heterogeneous sensor data management. In *Proceedings of the 5th international conference on Pervasive services*, ICPS '08, pages 121–130, New York, NY, USA, July 2008. ACM.

[42] Levent Gurgen, Claudia Roncancio, Cyril Labbé, Vincent Olive, and Didier Donsez. SStreaMWare : un intergiciel de gestion de flux de données de capteurs hétérogènes. In *23emes Journees Bases de Données Avancees (BDA'07) – Session démo*, October 2007.

[43] Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *International Workshop on Semantic Web Enabled Software Engineering (SWESE'06)*, Athens, USA, November 2006.

[44] Pat Hayes and Chris Menzel. Ikl specification document.

[45] Ian Horrocks, Peter F. Patel-Schneider, and Frank Van Harmelen. From shiq and rdf to owl : The making of a web ontology language. *Journal of Web Semantics*, 1 :2003, 2003.

[46] Jonathan W. Hui. *An Extended Internet Architecture for Low-Power Wireless Networks - Design and Implementation*. PhD thesis, EECS Department, University of California, Berkeleys, Sep 2008.

[47] IGN - Institut Géographique National. Géoportail, November 2011. http://www.geoportail.fr/.

[48] ISO/IEC 24707 :2007. Common logic (cl) : A framework for a family of logic-based languages, 2007.

[49] Paul Istoan, Gregory Nain, Gilles Perrouin, and Jean-Marc Jezequel. Dynamic software product lines for service-based systems. *Computer and Information Technology, International Conference on*, 2 :193–198, 2009.

[50] Maddy Janse, Fano Ramparany, Basilis Kladis, Leo Rozendaal, Tom Broens, and Henk Eertink. Specification of the amigo abstract system architecture. Technical report, IST Amigo Project, 2005.

[51] Thierry Jéron, Hervé Marchand, and Vlad Rusu. Symbolic Determinisation of Extended Automata. In *4th IFIP International Conference on Theoretical Computer Science*, volume 209/2006 of *IFIP International Federation for Information Processing*, pages 197–212, Stantiago, Chile, Chile, August 2006. Springer Science and Business Media.

[52] Gabriel Kalyon. *Supervisory Control Of Infinite State Systems Under Partial Observation*. PhD thesis, Université Libre de Bruxelles, 2010.

[53] Gabriel Kalyon, Tristan Le Gall, Hervé Marchand, and Thierry Massart. Control of Infinite Symbolic Transition Systems under Partial Observation. In *European Control Conference*, pages 1456–1462, Budapest, Hongrie, August 2009.

[54] D Kehagias. OASIS - Rapport D1.2.2 Wikipedia-like Web-based Tool for Framework Update and Maintenance, jan 2008.

[55] D Kehagias, S. Malasiotis, K Kalogirou, and I. Normann. OASIS - Rapport D1.3.2 Data Content Connector, Jan 2008.

[56] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. Mining software repositories with isparol and a software evolution ontology. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.

[57] Seong Hoon Kim, Jeong Seok Kang, Kwang Kook Lee, Hong Seong Park, Sung Ho Baeg, and Jea Han Park. A upnp-zigbee software bridge. In *ICCSA*, pages 346–359, 2007.

[58] Holger Knublauch. Ontology-driven software development in the context of the semantic web : An example scenario with protege/owl. In David S. Frankel, Elisa F. Kendall, and Deborah L. McGuinness, editors, *1st International Workshop on the Model-Driven Semantic Web (MDSW2004)*, 2004.

[59] Tristan Le Gall, Bertrand Jeannet, and Hervé Marchand. Supervisory Control of Infinite Symbolic Systems using Abstract Interpretation. In *44nd IEEE*

Conference on Decision and Control (CDC'05) and Control and European Control Conference ECC 2005, pages 31–35, Seville, Espagne, December 2005.

[60] Seok Won Lee and Robin A. Gandhi. Ontology-based active requirements engineering framework. In *in [Proc. 12th Asia-Pacific Soft. Engg Conf.], 481–490*, 2005.

[61] Nia-Chiang Liang, Ping-Chieh Chen, Tony Sun, Guang Yang, Ling-Jyh Chen, and Mario Gerla. Impact of node heterogeneity in zigbee mesh network routing. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 187–191, October 2006.

[62] Fouzia Meliouh and Kheira Tabet Aoul. L'habitat espaces et reperes conceptuels. *Courrier du savoir*, 1 :59–64, 2001.

[63] Riichiro Mizoguchi and Kouji Kozaki. Ontology engineering environments. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies : International Handbook on Information Systems*, pages 315–336. Springer, 2009.

[64] G. Nain, O. Barais, R. Fleurquin, and J.-M. Jézéquel. EntiMid : un middleware aux services de la maison. In *3ème Conférence Francophone sur les Architectures Logicielles (CAL'O9)*, Nancy, France, March 2009.

[65] OSGi. OSGi dynamic module system for java - service platform release 4, November 2010. http://www.osgi.org.

[66] M Panou and E Bekiaris. OASIS - Rapport D5.1.3 Project Presentation, jan 2007.

[67] A. Paz-Lopez, G. Varela, S. Vazquez-Rodriguez, J. A. Becerra, and R. J. Duro. Integrating Ambient Intelligence Technologies Using An Architectural Approach. In F. J. Villanueva Molina, editor, *Ambient Intellegence*. InTech, March 2010.

[68] Marina Petrova, Janne Riihijärvi, Petri Mähönen, and Saverio Labella. Performance study of IEEE 802.15.4 using measurements and simulations. In *Proceedings of IEEE WCNC*, April 2006.

[69] Christophe Roche. Terminologie et ontologie. *Larousse Revue*, Langages(157), Mar 2005.

[70] Vlad Rusu, Lydie Du Bousquet, and Thierry Jeron. An approach to symbolic test generation. In *In Proc. Integrated Formal Methods*, pages 338–357. Springer Verlag, 2000.

[71] Scala. http://www.scala-lang.org/, November 2011.

[72] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21 :96–101, May 2006.

[73] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet : A practical OWL-DL reasoner. *Web Semantics : Science, Services and Agents on the World Wide Web*, 5(2) :51–53, June 2007.

[74] Barry Smith. State University of New York at Buffalo, Buffalo Ontology Site, September 2011. http://ontology.buffalo.edu/.

[75] John F. Sowa. Conceptual graphs for a data base interface. *IBM J. Res. Dev.*, 20 :336–357, July 1976.

[76] Mani Srivastava, Richard Muntz, and Miodrag Potkonjak. Smart kindergarten : sensor-based wireless networks for smart developmental problem-solving environments. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 132–138, Rome, Italy, 2001. ACM.

[77] STG. http://www.irisa.fr/prive/ployette/stg-doc/stg-web.html, November 2011.

[78] Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp*, 2004.

[79] York Sure, Steffen Staab, and Rudi Studer. *Ontology Engineering Methodology*, chapter Ontology Engineering Methodology, pages 135–152. Springer, 2009.

[80] Mohammad-Reza Tazari, Francesco Furfari, Juan-Pablo Lázaro Ramos, and Erina Ferro. The PERSONA service platform for AAL spaces. In *Handbook of Ambient Intelligence and Smart Environments*, pages 1171–1199. Nakashima, Hideyuki and Aghajan, Hamid and Augusto, Juan Carlos, 2010.

[81] Phil Tetlow, Jeff Z. Pan, Daniel Oberle, Evan Wallace, Michael Uschold, and Elisa Kendall. Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering. W3C Working Draft Working Group Note 2006/02/11, W3C, 03 2006.

[82] Mike Uschold and Michael Gruninger. Ontologies : Principles, methods and applications. *Knowledge Engineering Review*, 11 :93–136, 1996.

[83] W3C. State Chart XML (SCXML) : State Machine Notation for Control Abstraction, april 2011. http ://www.w3.org/TR/2011/WD-scxml-20110426/ last access : 12/12/2011.

[84] Yimin Wang, Johanna Völker, and Peter Haase. Towards semi-automatic ontology building supported by large-scale knowledge acquisition. In *AAAI Fall Symposium On Semantic Web for Collaborative Knowledge Acquisition*, pages 70–77. AAAI, 2006.

[85] Mark Weiser. The computer for the 21st century. In *Human-computer interaction : toward the year 2000*, pages 933–940. Morgan Kaufmann Publishers Inc., 1995.

[86] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proc. First International Workshop on Semantic Web and Databases*, 2003.

[87] ZigBee Alliance web site. http://www.zigbee.org, November 2011.