Discrete Control in the Internet of things and Smart Environments through a Shared Infrastructure

Mengxuan Zhao

UNIVERSITÉ DE GRENOBLE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Mengxuan ZHAO

Thèse dirigée par **Hassane ALLA**
et codirigée par **Eric RUTTEN** et **Gilles PRIVAT**

préparée au sein du
Orange Labs, **Gipsa Lab** et **INRIA Rhône-Alpes**
et de **L'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Discrete Control in the Internet of Things and Smart Environments through a Shared Infrastructure

Thèse soutenue publiquement le **7 mai 2015**,
devant le jury composé de:

Eric NIEL
Professeur, INSA Lyon, Rapporteur

Thierry MONTEIL
Maître de conférence (HDR), INSA Toulouse, Rapporteur

Frédérique BIENNIER
Professeur, INSA Lyon, Examinatrice

Noureddine ZERHOUNI
Professeur, ENS2M, Examinateur

Hassane ALLA
Professeur, Université Joseph Fourier, Directeur de thèse

Eric RUTTEN
Chargé de recherche, INRIA Rhône-Alpes, CoDirecteur de thèse

Gilles PRIVAT
Ingénieur de recherche, Orange Labs, Co-encadrant de thèse

UNIVERSITÉ DE GRENOBLE

MSTII

Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

T H È S E

pour obtenir le titre de

docteur en sciences

de l'Université de Grenoble

Mention : Informatique

Présentée et soutenue par

Mengxuan ZHAO

Contrôle Discret pour l'Internet des Objets et les Environnements Intelligents au travers d'une infrastructure partagée

Thèse dirigée par Mengxuan ZHAO

préparée au laboratoire
Orange Labs, Gipsa-Lab et INRIA Rhône-Alpes

soutenue le date de soutenance

Jury :

Rapporteurs :	Eric NIEL	-	INSA Lyon
	Thierry MONTEIL	-	INSA Toulouse
Examinateur :	Frédérique BIENNIER	-	INSA Lyon
	Noureddine ZERHOUNI	-	ENS2M
Directeur :	Hassane ALLA	-	Université Joseph Fourier
CoDirecteur :	Eric RUTTEN	-	INRIA Rhône-Aples
Encadrant :	Gilles PRIVAT	-	Orange Labs

Abstract

The Internet of Things (IoT) and Smart Environments (SE) have attracted a lot of research and development activities during the last decade. Yet many present-day IoT/SE applications are still limited to the acquisition and processing of sensor data and its context, with control, if any, using either basic solutions or requiring human intervention, far away from the automatic control which is an essential factor to promote the technologies. This thesis targets to bring knowhow from control theory and reactive systems to the IoT/SE domain to achieve a solution with a formal method for the missing control aspect.

We propose the extension of a framework in order to build a shared generic IoT/SE infrastructure offering high-level interfaces to reduce design effort, and enabling the self-configuration and adaptation of control applications over generic properties of the environment without human interaction by using general knowledge over the domain that applies to each target instance of IoT/SE system. In this extended framework, individual physical entities (including all relevant "things", appliances and subsets of space) may be grouped as virtual entities by shared properties to provide a higher level abstraction for control and other applications and better adaptation to lower level configuration changes.

Requiring a generic common denominator solution shared by all IoT/SE applications in a given environment, we propose for this infrastructure, to model by finite state automata the target entities to be monitored and controlled, including both individual entities and their groupings, as well as things and space entities, to be able to apply discrete controller synthesis (DCS) technique over any of these at different levels of abstraction and granularity. DCS is a formal method which constructs automatically a controller, if it exists, guaranteeing the required control objectives regarding to the given system behavior model in terms of synchronous parallel automata. The existing BZR programming language and Sigali tools are employed to perform DCS and generate a controller in an automatic way.

Necessary supporting software modules are proposed in the implementation such as the relation maintenance module keeping the correct association between individual entity instances and groups, and dispatching the action orders from the high level control to corresponding actuators. This module would evolve later to a more generic solution such as a graph data base including both the general knowledge base and specific environment instance relations. Conflict resolution between objectives of control coming from concurrent controllers is also indispensable due to the intended openness of the platform. A java based context simulator has been developed to simulate the home environment within several scenarios proposed for the validation, such as electrical load control and activity context adaptation.

L'Internet des Objets (IdO) et les Environnements Intelligents (EI) ont attiré beaucoup d'activités de recherche et développement au cours de la dernière décennie. Pourtant, de nombreuses applications IdO/EI d'aujourd'hui sont encore limitées à l'acquisition et au traitement des données de capteurs et de leur contexte, avec un contrôle, le cas échéant, utilisant soit des solutions de base ou demandant l'intervention humaine, loin du contrôle automatique qui est un facteur essentiel de promouvoir ces technologies. Cette thèse vise à apporter le savoir-faire de la théorie du contrôle et des systèmes réactifs dans le domaine IdO/EI pour arriver à une solution avec une méthode formelle pour l'aspect de contrôle qui fait défaut.

Nous proposons l'extension d'un canevas logiciel pour une infrastructure générique et partagée IdO/EI qui offre des interfaces de haut niveau pour réduire l'effort de conception, et qui permet l'auto-configuration et l'adaptation des applications de contrôle sur des propriétés génériques de l'environnement sans intervention humaine en utilisant les connaissances générales sur le domaine qui s'appliquent à chaque instance cible de système IdO/EI. Dans cette infrastructure étendue, les entités physiques individuelles (y compris toutes les "choses", appareils électriques et sous-ensembles de l'espace) peuvent être regroupées comme des entités virtuelles par des propriétés communes afin de fournir un niveau d'abstraction plus élevé pour le contrôle et d'autres applications, ainsi qu'une meilleure adaptation aux changements des configurations au niveau inférieur.

Sur le requis d'une solution générique et commun dénominateur partagée par toutes les applications de l'IdO/EI dans un environnement donné, nous proposons pour cette infrastructure, de modéliser les entités cibles supervisées et contrôlées, y compris les entités individuelles et de leurs regroupements, ainsi que les choses et les entités spatiales, par des automates à états finis, pour être en mesure d'appliquer la technique de la synthèse des contrôleur discrets (SCD) aux différents niveaux d'abstraction et de granularité. SCD est une méthode formelle qui construit automatiquement un contrôleur, s'il existe, en assurant les objectifs de contrôle exigés concernant le modèle de comportement du système donné en termes d'automates parallèles synchrones. Les langages de programmation BZR et les outils Sigali existants sont utilisés pour effectuer la SCD et de générer un contrôleur de manière automatique.

Les modules logiciels nécessaires sont proposés dans l'implémentation tels que le module de maintenance de relation qui garde une association correcte entre les instances d'entités individuelles et les groupes, et répercute des commandes d'action du contrôle de haut niveau aux actionneurs correspondants. Ce module est destiné à évoluer plus tard vers une solution plus générique comme une base de données graphes comprenant à la fois la base de connaissances générales et relations spécifiques d'instance environnement. La résolution des conflits entre les objectifs de contrôle venant de contrôleurs concurrent est également indispensable en raison des objectifs de l'ouverture de la plateforme. Un simulateur de contexte basé sur Java a été développé pour simuler l'environnement de la maison au sein de plusieurs scénarios proposés pour la validation, tels que le contrôle de la charge électrique et l'adaptation au contexte de l'activité.

Contents

List of Figures

List of Tables

List of Acronyms

ABM Agent-based modeling. 28, 37

API Application Programming Interface. 16, 43

CPS Cyber-Physical System. 37

CRUD Create, Read, Update, Delete. 17

DAG Directed Acyclic Graph. 3, 37, 47, 61, 75

DCS Discrete Controller Synthesis. 34, 37, 59

DES Discrete Event System. 29, 33, 37

DNS Domain Name System. 12, 24

FPGA Field-Programmable Gate Array. 36

FSM Finite State Machine. 29, 31, 37, 45

HVAC heating, ventilating, and air conditioning. 20

ICT Information and communications technology. 1, 10, 13, 44

ITU International Telecommunication Union. 24

M2M Machine to Machine. 10, 24

OOM Object-Oriented Modeling. 27, 37, 55

OSI Open System Interconnection. 14

QoS Quality of Service. 20

RDF Resource Description Framework. 22, 45

REST REpresentational State Transfer. 10, 44, 89

RFID Radio Frequency IDentification. 2, 8, 13, 44

ROA Resource-Oriented Architecture. 54

SOA Service-Oriented Architecture. 8

UPnP Universal Plug and Play. 2, 8, 13

URI Uniform Resource Identifier. 12, 21

W3C World Wide Web Consortium. 21

WSAN Wireless Sensor Actuator Network. 14, 25

Chapter 1

Introduction

Contents

1.1 Scope Identification

Smart Environments (SE) are meant to enhance the interaction between humans and the physical world in which intelligence is embedded seamlessly through sensor networks and ICT devices. They have mostly been seen as an extension and evolution of traditional device-based human interfaces, using generic models of space. On the other hand, the Internet of Things (IoT) aims to integrate a large number of everyday things, including non-ICT objects, into the digital world, to make them communicate and interact, thanks to unique addressing and communication technologies. With an extremely large application spectrum from supply chain management to transportation assistance, from health care to social network, the IoT offers great potentialities to development of new services and applications, in order to not only improve the everyday life of private users, but also to make benefits to the whole society by increasing the efficiency and productivity in industrial and public domain. However, it is unfortunately too large to come up with a generalization over the entire range.

Therefore, the scope of the current thesis is limited to the intersection of the IoT and Smart Environment, i.e. the IoT restricted to the application domain of Smart Environment, including Smart Home, Smart building, Smart City, as a trade-off of the genericity and the usefulness of the proposition in the current thesis. Thanks to the lower and lower cost, network connectivity equipment becomes more and more accessible to the public, which makes this intersection of the two domains booming in terms of number of applications, number of industrial investigators and amount of research interest. They have some shared characteristics which our proposition will take into account: *openness*, *dynamicity*, *heterogeneity*, *non reproducibility* and *similarity* from one instance to another. In spite of their common points, the nuance between the IoT and SE makes the practiced research approaches in the two domains different: in the IoT, a "thing" vision is more adopted while in SE, it is more environment centric. Very often, to resolve a problem involving both

"things" in the IoT and "environment" in SE, at least a distinct "thing" model and an "environment" model are given. This makes the control designing even more complicated which consists of cutting the problem into two parts to be modeled separately and controlling the two parts in a consistent way. Thus, the second reason we choose this intersection to make contribution is that we would like to propose a uniform modeling method within the scope to reduce the complexity of modeling procedure.

In the entire document, unless explicitly stated, we try to make our proposition generalizable to the underlined scope thus defined which is referred to as the *target scope* or *IoT/SE*.

1.2 Problem statement

Many present-day IoT/SE applications are still limited to unidirectional data collection from sensors for remote monitoring purposes with remote control, if any, using either basic solutions relying on unquestioned assumptions or requiring human interaction in a *human-in-the-loop* fashion, far away from the automatic control which is an essential factor to make technologies invisible. Besides, they often get limited to specific identification technologies, such as RFID, UPnP, and network technologies such as lower-power radio. The automatic control aspect is generally missing in such applications. Two main causes responsible for this situation are identified:

- First, concerning the classic control in industrial domain, mainstream applications still rely on fully customized and ad hoc top-down vertical design laying directly on the physical interface, i.e. sensor and actuator level, which has been so far dominant in classic automatic control and cyber-physical systems. Control applications in such case need to be installed and configured by an expert capable to specify the individual correspondent devices to each application in each individual environment which is a heavy and tedious procedure and not supposed to be within the ability of the public. If any configuration change takes place during the run time, which is unfortunately quite frequent due to the highly dynamic nature of such environment instances, there is also need of expertise. This individual design and installation, plus the maintenance expertise makes the total cost of such application extremely high which is unacceptable to the constraints of mass-market in the IoT/SE domain.
- Second, concerning the emerging consumer SE platforms, weakness of some present day rule engines for control applications is observed: sequential execution of rules may create random non-expected results if the order of rules is altered, and no guarantee of objective achieved if the open loop control approach is adopted. Moreover, the best-effort and time-insensitive culture inherited from the general information engineering by generic infrastructures in the IoT/SE domain, which could be disturbing in some reactive applications with time delay constraints. The lack of formal method makes the reliability and effectiveness of such control applications not satisfactory.

To release the entire potential of the promising IoT/SE domain, automatic control which is an important way to make the environment intelligent, should be more accessible

to the public in terms of cost and liability. In order to reach this goal, we argue that the following elements are inevitable to overcome the above issues:

- A shared infrastructure offering high-level interfaces to reduce design effort, and enabling the self configuration and adaptation of control application on generic properties of the environment without human interaction by using general knowledge over the domain and creating horizontal layers according to the abstraction levels of the physical world.
- Application of new control techniques, possibly from more critical domain such as real-time embedded systems, on the above proposed infrastructure in the IoT/SE domain to deal with the liability of desired result and real-time issues present in many today's control applications.

1.3 Contributions

In this thesis, we present several contributions to the previous objectives:

We begin with the proposition of a high-level framework using a knowledge base over the domain and a method of modeling in order to integrate target physical entities into the ICT system. The invariant knowledge base over different environments of the domain, also called *domain-specific ontology*, structured in a directed acyclic graph (DAG), allows to capture the generic properties over the domain and arrange them following a hierarchical relationship, which makes it easier to make variable abstraction level of the target environment according to the demanded need of the given application. Each node of the DAG, representing either a *property* or a concrete object *type*, is provided with a finite state model which is used as a template during the auto discovery phase, or as instantiation blueprint for individual objects. Environment entities are presented in the same way as "things" by a finite state model and occupies a parallel branch to all the "things". The adaptability over various environment instances is furthermore enhanced by making the change at physical entity level transparent to high-level applications. The approach is to model the target group of objects sharing the model at the abstraction level required by the application, i.e. the node in the DAG that the application addresses. This *group model*, possibly specific to each generic application requirements but available in all environment instances, allows the reuse of these applications without dozens of device-application input/output associations to do by offering a unified and invariant high-level interface.

To respond to control requirements, we propose a discrete control approach based on a tool-supported synchronous variant. It favors automatic and correct-by-construction manager derivation which has been applied in diverse domains. Models of relevant groups of entities are considered as synchronous parallel automata on which the discrete controller synthesis (DCS) is performed with the desired objectives expressed formally as logical equations. A controller is generated if there is a solution, which warrants the objectives are satisfied. All necessary supporting software modules are also proposed such as the topology maintenance module keeping the correct association between individual entity instances and groups of entities and dispatching the action order from the high level control

to corresponding actuators. Conflict resolution between different objectives of control is also indispensable due to the openness of the domain where objectives are from diverse origins.

1.4 Thesis outline

This thesis is divided into three parts.

Part I presents the state-of-the-art, interleaving background with related work. It has two chapters. Chapter 2 identifies the common characteristics, the current standards to which our proposition would made contribution, the enabling technologies as a start point of our proposition and the most popular applications already available on the market in our target scope. Chapter 3 presents the existing models, languages and tools that can be employed to address the targeted design issues. Among them, we identify and introduce the synchronous models, and discrete control techniques and tools that will be applied in the thesis.

Part II exhibits the contributions of the thesis. It has two chapters. Chapter 4 first presents an overview of our proposed framework. The focus is put on the modeling aspect of "thing", "space" in the target scope, as well as the modeling of an abstract type or a shared property by finite state machine. Their hierarchical relationship should be correctly established and a group of same type entities is possibly abstracted by one model to reduce the complexity of the system. Chapter 5 presents the mechanism implemented in the proposed framework to enable to use discrete control techniques, and the conflict resolution over generic control rules taken from a shared corpus.

Part III describes the implementation of such framework extending an existing one on OSGi. Several case studies using the framework are presented, as well as some discussions over their results.

Finally, Chapter 8 concludes concludes the thesis and presents some outlook.

Part I

State of the art

Chapter 2

General background of the IoT and SE

Contents

This chapter describes the general context and background of the scope of the present thesis. We begin with the identification of common properties of the addressed domain which will serve as a base of requirements to our proposition. Current standardization and reference model effort is then presented. Commonly used enabling technologies are indispensable to be considered in today's researches and applications. The application scope we are aiming is extremely large where research directions are numerous and various. We hope to make a clear position of our proposition through this chapter.

2.1 Common characteristics and challenges

The target scope is a particular hardware and software environment. This section describes the identified shared properties and the challenges raised by each of them.

Openness. Devices in an instance of IoT/SE environment, including sensors/actuators and smart objects, are often made by different manufacturers. Out-sourced data and information are from different providers. Applications that give intelligence to the environment along with the devices and data are also developed by different developers. One environment instance is not necessarily managed by only one stakeholder, which is especially true in building and city scales. The openness enables the data sharing among objects, devices, applications and systems, which helps improving the artificial intelligence, for example the accuracy of automated decisions based on the context information, and the accuracy of pattern recognition. It promotes also the development of such applications. This *openness* requires the environment modeled as a comprehensive system, supporting different applications, rather than one by one in a top-down fashion as a set of vertically integrated solutions for individual applications.

Dynamicity. An IoT/SE environment is supposed to be *dynamic* during its lifetime, as devices appear and disappear all the time and some of them can move within the environment, context information is changing permanently on runtime. The entire system should be able to detect the changes in the environment, to adapt itself to the new configuration and to act correctly to keep the whole system respecting current requirements. The latency introduced by the whole reaction chain should respect the *temporal constraints* that prescribed by individual applications, i.e. a lot more severe for real time applications than for time-insensible ones like the update of meteorological information.

Heterogeneity. The devices and applications in an IoT/SE environment provide various functionalities. Different devices and applications may offer the same functionality. They use different communication protocols, like Wifi, Zigbee, ZWave for indoor wireless communication, and they may be connected by diverse identification technologies like RFID, UPnP. Apart from the objects coming with a network connection and some intelligence, the so-called *Smart Objects*, there are things of previous generations which do not have an embedded network connection. To cope with this *heterogeneity*, there should be a universal and generic model to capture the invariants from the diversity in a *semantic* way in order to represent them independently of the technologies or vendors. The *SOA* allows a loose couple between applications and devices to hide the implementation details of a given functionality or assures a feature faced to the *dynamicity*.

Non reproducibility. Industrial control applications had manufacturing, dedicated systems (example: avionics), people with specified needs as their targets in the past. The common point among those applications is that the design is a case-by-case one and the costs of such design are recovered by producing a large number of identical items. Unfortu-

nately in the target scope, almost non of the application instances is identical to another, even in a block of identically built houses where the inside installation is different according to the house holder's need. By contrast to the solutions like luxury residence, production line which do not have much constraint on cost, an entire case-by-case design does not fit for the solutions because of its high cost of design and maintenance (money and time) and the non reproducibility of those instances, in which a self-configurable solution is much more attractive.

Similarity. Though all instances of IoT/SE environments are rarely identically reproducible, they are structured by a few broad organizational features that are invariant from one instance to another: all homes are composed by rooms, all buildings are divided into floors and rooms, all cities are structured by streets, crossings, blocks, etc. The physical entities in these environments are also similar from one instance to another like all homes have some generic categories of appliances such as washing machine, lamps, heatings, etc. This characteristics makes sense of a general solution that can self-configure once installed in an environment with minimum human input based on generic and reusable models of entities in these similar environment instances.

2.2 Current standards and references

Facing the above characteristics, especially the *openness* and *heterogeneity*, standards are urgently needed to establish a baseline or a common grounding to improve high-level interoperability. According to the fact that different domains have specific needs, various standards may exist with higher or lower level abstraction targeting more or less generic use cases: home, city, personal health, industry, etc. It is an ongoing work with a lot of activities. Standardization organizations, companies, academic institutions, governments are all involved and contribute their propositions.

2.2.1 ITU-T recommendation Y.2060: standard for the IoT in general

ITU (International Telecommunication Union) is the United Nations specialized agency for information and communication technologies – ICTs[1]. Its Standardization sector (ITU-T) develops international standards known as ITU-T Recommendations, some well known examples are JPEG for imaging coding, H.264/MPEG-4 for video coding, (x)DSL for internet access via a telephone line.

The current Y.2060 provides an overview of the IoT by giving a brief introduction of the IoT concept, identifying its common characteristics and high-level requirements in order to establish the IoT reference model (in figure 2.1) which has 4 horizontal layers together with 2 transversal supporting modules corresponding to identified requirements arranged from the closest to software application to the closest to physical things. This very high level and general reference model will be the basis of our proposition of architecture.

[1] http://www.itu.int/

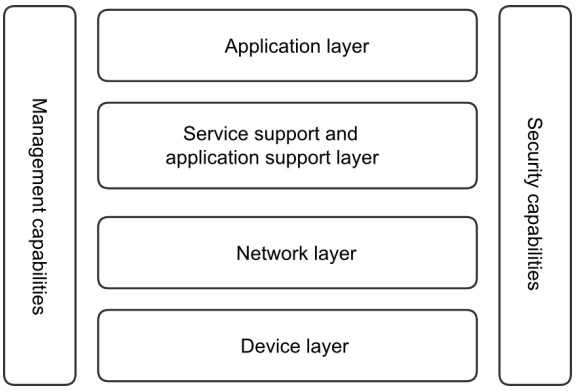


Figure 2.1 – The IoT reference model of ITU

While standards for IoT in general are kept quite abstract to enable different concrete architectures, technologies responding specific requirements coming from the broad application domains that general IoT covers, more specific standards are also under work to provide more practical and concrete reference and guidelines.

2.2.2 ETSI M2M: Identification & communication oriented

The term M2M takes its origin in the telecom industry as firstly an extension of services using the cellular network to connect terminal with embedded SIM cards. M2M services are now understood as potentially using all kinds of networking and technology to enable data exchange among machines and considered having a large range of applications. ETSI M2M[2] has have published standards for from overall functional architecture to detailed communication interfaces, together with semantic models and use case identification. A large part of ETSI M2M standards have been taken by OneM2M[3], a global organization developing standards for M2M communications and the IoT, which encompasses ETSI and other organizations as active partners.

ETSI M2M standardization aims to provide a *horizontal* Service Capability Layer (SCL) which is a set of services accessible from shared and open interfaces: instead of a *vertical* ICT infrastructure developed by each vendor for its proper applications, a shared service and network platform is provided which takes care of the supportive functionalities so that developers or users only have to declare the objects in a standardized way and use the common interfaces to implement the applications. The entire architecture is divided into 3 main domains: network, gateway and device, which interact among themselves via mId interface and interact with applications with mIa or dIa interfaces. These interfaces are generic and extensible. It adopts a RESTful architectural style (2.3.4.1) in the sense that each SCL contains a standardized resource tree where each resource is accessible via a URI by standard CRUD (Create, Read, Update, Delete) requests.

OM2M [Ala+14] is an open source M2M service platform which implements the ETSI M2M standards, especially the part over the functional architecture and communication

[2] http://www.etsi.org/technologies-clusters/technologies/m2m
[3] http://www.onem2m.org

interfaces, for the first time. It is built on top of the OSGi Equinox[4] runtime adopting a modular architecture that features or functions of an SCL specified in the standards are implemented as a "plugin" to facilitate the SCL extension as required in the standard. Services are exposed via RESTful API and an object oriented database is provided. This implementation provides also a self-configuration mechanism using autonomic computing plugin, which facilitates the despoilment of a M2M network.

The present OM2M solution's advantage is that it is an implementation of a well known standard ETSI M2M which makes it easy to be accepted by application developers and compatible with other M2M solutions. However, ETSI M2M defines services and interfaces addressing individual devices such as sensors, smart or legacy devices without recommendations on how to manage these devices and how to use data they provide. A pre-normative study on semantics for M2M data is ongoing. Therefore, OM2M implementation does not provide any support for M2M data processing apart from the autonomic self-configuration mechanism following basic SWRL (Semantic Web Rule Language) rules to bind sensor/actuator to monitor/controller. Application developers should at their own charge to define the signification behind each individual resource and use them properly.

2.2.3 Smart Home Environments

The two following are either reference model or pre-standardization activity which are susceptible to be a standard in the future .

HGI (Home Gateway Initiative)[5] is an industrial alliance of service providers targeting the digital home aims to provide a *flexible Smart Home delivery platform* connecting home devices and the internet where services from various vendors can be deployed easily and interoperate. It has published reference Smart Home architecture, requirements for network connection and service support collected from members. It is also a partner of OneM2M.

IEEE-SA IC, Convergence of Smart Home and Building Architectures. Both the similarities among Smart home and building architectures and a gap preventing the internetworking between these domains have been observed during the latest years. This ongoing per-normative activity[6] aims to offer a seamless experience over multiple domains by converging the Smart Home and Building architectures to enable inter-domain internetworking which creates benefits in many aspects such as increased granularity, visibility and awareness, increased level od usability, etc.

Smart Appliances[7] is a project trying to result a reference ontology that fits the ETSI M2M architecture, with the scope of the consumer market of the home, public buildings

[4] http://www.eclipse.org/equinox/
[5] http://www.homegatewayinitiative.org
[6] http://standards.ieee.org/develop/indconn/activities.html
[7] https://sites.google.com/site/smartappliancesproject/home

and offices, as well as the standard appliances in such environment. It has studied a range of existing semantic assets and use case assets to be translated and mapped to make up a common reference ontology, called "Smart Appliances REFerence (SAREF) ontology" (c.f.2.4.2). This resulting ontology is to be contributed to ETSI as a future standard.

2.2.4 Naming/Identification standards

An identifier which does not create confusion in a reasonable range is essential for managing things in the IoT/SE scope. According to the nature of different application domains, requirements and approaches can be different. Scalability, searchability and the capability to provide basic configuration information are some identified requirements for such standards.

Domain Name System (DNS) is a hierarchical distributed naming system to identify resources connected to the Internet or any network, standardized as an RFC (Request for Comments) document published by IETF (Internet Engineering Task Force)[8]. It translates easily memorized domain names to IP addresses needed for the purpose of locating services and devices. The domain names are arranged in a tree structure, the Domain Name Structure, that reflects in the syntax that we see in every single name like www.example.com. There are components for registering, storing and resolving domain names like a database. It can be used in the domain of IoT/SE for finding the knowledge data for a given entity by the URI associated to it.

Electronic Product Code (EPC)[9] is a universal identifier that gives to a physical object an identity unique among all objects. It was first build in the context of supply chain item tracing and management, but expanded and gained its popularity thanks to RFID in which the EPCs are encoded in most of the cases. The main idea of EPC is to attribute each object a globally unique code that two identical objects can be distinguished. In an EPC identifier, numerous information can be found, such as information about the EPC code (generation, length), manufacturer, object class and unique serial number. These informations are accessible via several services coming with the EPCglobal Architecture Framework (EPCAF), such as the Object Name Service (ONS) which is an automated networking service similar to the DNS.

2.3 Technology background

The IoT or Smart Environments would not been existing without any of the following technologies. The development of IoT qnd SE has also promoted the research and development of these technologies. This technology background will establish the base and the start line of our research.

[8] available on http://www.rfc-editor.org/
[9] http://www.epc-rfid.info/

2.3.1 Identification and configuration

Things should be first identified in order to be integrated into the network to provide data and get control. Nowadays, most identification technologies consist of a *naming*, *addressing* or *tagging* phase as a prerequisite, either during manufacturing like attributing a MAC address for classical network devices, embedding a SIM card in a telco style, or giving attached tag later like Barcode, RFID tag. Unfortunately this *tagging* process can be very costly and tedious since the number of "things" is potentially in the order of trillion. Automatic identification and data capture technologies based on nature features like using biometric data are under exploitation [CAS] to extend the IoT which has been originally defined as the "global networking connecting any smart object" by the IoT-A project[10] to all *sense-able* things without necessary a digital interface [Pri12] like a legacy appliance or a human.

The identification is not the final objective. After this phase has been accomplished, the *configuration* process takes place to integrate the newly identified object into the network to enable monitoring and control over it through the discovered interface. The plug&play basis which assumes "zero configuration" discovery and integration of new devices is now adopted in many network and distributed infrastructures. The self-configuration capability may be in low level like the automatically assigned private network IP address by DHCP (Dynamic Host Configuration Protocol) when a new device with network interface arrives, or in a higher level, such as in UPnP where networked devices are assigned by a unique address, advertise the services it supports to get discovered. After the discovery, the descriptive in XML format coming together with the device details its information and capabilities to be get controlled by the system.

However, these solutions are limited to digital things with a network-ready interface or tag. What's more, devices should be fully identified to very specific types to get things to work. An approximate identification of a general category like kettle, TV is not enough to be integrated in a regular Service oriented architecture. In fact, it is often sufficient for applications to work with an approximate identified thing to get things speed up, as long as the basic required features are available in the approximate type. It should also be noted that the fashionable unique ID does not provide metadata such as the knowledge of the context, which brings an other challenge that useful contextual data should be carefully and efficiently associated with the ID.

In [Hu14] the authors propose an approximate and contextual identification and configuration mechanism. The target entities are physical things or subsets of space without a native network interface. They are the *sense-able* and *actionable* (if possible) objects through the intermediate of sensors and actuators playing the role of network interface. Target entities do not need to be tagged in advance, instead, they are to be discovered: models of generic categories (like "printer") are available in the system as identification templates with which the classification algorithm, like energy consumption pattern recognition, compares the available sensor data to determine the type of the entity to be integrated. An ICT *shadow*[11] is automatically created as representative of the target physical entity

[10] www.iot-a.eu

[11] We will take this notion of *ICT-shadow* renamed as *proxy* in our proposition

for the ICT system which is capable of self-configuring, monitoring and controlling these entities. The advantages of this proposition are (a) no need of *digitization* for everything to get integrated in the network; (b) no need of complete identification: a very generic model will be given to the discovered entity until more available data enabling a finer and precise identification as a *re-configuration* process.

2.3.2 Wireless sensor actuator network

A wireless sensor network (WSN) consists of a certain number of sensors to monitor the physical environment and pass their measurement data to where they are needed through the network. With the decreasing price of small and low-power sensors thanks to the significant progress of micro-electro-mechanical systems (MEMS) technology in recent years, WSNs, which become WS**A**Ns including actuators, have promoted and become a basic enabler and crucial part of the IoT and Smart Environment.

As the **N** its names reveals, a WSAN is at first a *network* which has similar characteristics than other classical networks like IP network, as well as open issues. In a survey over the WSN [Aky+02], a communication architecture of sensor networks is presented where individual *sensor nodes* with data collecting and routing capabilities form a *sensor field* connected to the Internet through a *sink* playing the role of a gateway. The presented protocol stack with 5 layers is very similar to the OSI reference stack model. Common issues including:

- Scalability. The number of potential sensors and actuators can be very large like a person can carry hundreds of sensor nodes embedded in glasses, clothes, shoes, watch, etc. [Aky+02], the current addressing system, mostly based on the IEEE 802.15.4 standard, may fail to identify, integrate and manage in a efficient way. Research directions includes semantic approaches, new addressing systems like IPv6 addressing been proposed within the 6LoWPAN [SB11] context.
- Security. Security has been widely studied in all traditional computing and communication systems, yet it remains a study topic and the WSN is extremely vulnerable to attacks as (a) its wireless communication making eavesdropping very simple; (b) its nodes are with limited in terms of both energy and computing capabilities that they cannot support complex scheme of security like complex encrypting or authentication process. Current standards or technologies mentioned earlier in the chapter all take this problem into account and have a dedicated module for the security, as well as for privacy.

WSANs also have some particularities due to its nature:

- Power consumption constraint. Depending on the application, WSANs can be very different in size, in fault tolerance, in accessibility, etc. from one to the other. In any case, changing battery would be an expensive operation considering the total number of sensor/actuator nodes, the embedded aspect, or even impossible like the WSAN for battlefield surveillance, meteorological inspection, tracking of movement of animals,

etc, which infers that the life time of such network may depend on the power resource of the component devices. Thus an important requirement on such devices is that they work at extremely low power du to the limitation in their (battery) size.

- Limited computational capabilities. Though technological progress is making higher and higher computational power in smaller and smaller unit, processing and memory are still precious resources in sensor/actuator nodes. For example, in a Texas Instruments system-on-chip solution for Zigbee, the processing unit is a 8051 microcontroller with up to 256 KB Flash memory and 8 KB of RAM. Therefore, these capabilities are needed for sensing, actuating, communication and other data processing objectives to interact with their surroundings. This feature should limit the data processing rate for the devices.

Facing these issues, protocols designed specifically for WSAN are many according to various application domains. For example, Wireless HART (Highway Addressable Remote Transducer)[12] protocol is very popular in industrial implementations for its capability of communicating over legacy analog instrumentation wiring, while Bluetooth is widely used for communications between personal terminals. Zigbee[13] is a suite of of high level communication protocols based on an IEEE 802.15 standard, the same as mentioned Wireless HART and 6LoWPAN. It is a low-rate and short-distance technology which meets the requirements of WSAN. It is mostly suitable for Wireless Personal Area Network (WPAN) as its range is 10–100 meters line-of-sight. Z-Wave[14] is also an extensively used WPAN protocol built upon ITU-T G.9959 specification, sharing the same position as the 802.15 family.

2.3.3 Sensor data aggregation and complex event processing

Sensor data aggregation consists of a preliminary data processing of low level information, often the raw data directly from sensors. This processing aims to provide more accurate and meaningful information about the physical world without complex data processing algorithms or extra knowledge. A lot of work has been done in this research direction, here are some examples. In the work of [Tap+06], a flexible kit of wireless sensing devices for pervasive computing research in natural settings is developed. In [Gur+08], a service-oriented middleware is put forward for heterogeneous sensor data aggregation and management. The authors of [TIL04] propose an aggregation of raw data coming from either sensors or other sources in order to recognize activities in home areas. [LF09] uses data from a variety of wireless sensors to build up a robust location-aware activity recognition for smart home applications.

By contrast to sensor data aggregation, the complex event processing which takes multiple continuous and discrete data from on-site sensors to compute a result with the help of external data about the environment. Among numerous FIWARE generic enablers (GE), a CEP (Complex Event Processing) GE[15] analyses event data in real-time, reacts to situ-

[12] http://en.hartcomm.org/
[13] http://www.zigbee.org
[14] http://www.z-wavealliance.org/
[15] http://catalogue.fiware.org/enablers/complex-event-processing-cep-proactive-technology-online

ations based on a series of events that have occurred within a dynamic time window called processing context, rather than to single events. It is implemented by the Proactive Technology Online, a scalable integrated platform providing tools and web user interface for performing the CEP GE.

The two methods can do their job individually, or be combined in a given context in order to realize the context-awareness for a specific application.

2.3.4 Middleware

Sensors and actuators come from various vendors, using different protocols and are not necessarily interconnected. To be truly useful, they need to be ubiquitously and seamlessly integrated into the network. On the other hand, application programmers are not assumed expert in sensor and actuator protocol to be able to develop applications. Thus, an intermediate layer or several sub layers are needed between the device and the application layers to provide a unifying platform. This layer or sub layers, defined as middleware, should hide technological details, provide abstraction of things, useful services and high-level APIs to applications such as interoperation at network, syntactic and semantic level, basic data aggregation, resource discovery, security management, available command on the physical things [Ban+11]. As the application development become more and more rapid and devices become more and more divers, the middleware has become more and more indispensable for simplifying development of applications using heterogeneous devices. It enhances the separation of concerns by horizontalizing the infrastructure. Programmers no longer need to have the exact knowledge of the technologies adopted in the low level and develop various versions for the same application to adapt the low level.

2.3.4.1 System architecture: SOA vs ROA

In the current literature, the *Service Oriented Architecture* approach is widely adopted by middleware architectures in IoT and Smart Environments [Spi+09]; [VM11]; [Mar+13]; [Tei+11]. It is based on *services* which are units of functionality and can be consumed by client not necessarily from the same vendor or using the same protocol. SOA gains its popularity for several reasons. First, it decouples the *service consumer* (client) and *service provider*. Secondly, interaction is done by simple messages over common interfaces and standard protocols instead of whole object reference and data exchange which is specific to implementation and may be influenced by network performance. At last, a significant advantage of SOA is that it allows software and hardware reusing as it does not impose a specific technology for the service implementation as long as the services respect the required interface for the clients. *Resource Oriented Architecture (ROA)* is an other architectural style as an evolution of SOA after some weakness observed in SOA such as pool performance of message translation due to heavy data structure of XML and the need of knowledge of the contract of service before using it [Ove07]. Instead of transferring a SOAP message with a heavy overhead like in a typical SOA, ROA makes every entity a resource and interactions are around the *stateless* representations of the resources [Fie00]. Here, the signification of *stateless* is that a system state representation responding to a request

is independent of the previous system state. Unlike SOA which has several possibilities in choice of transport protocols, including HTTP, ROA is built upon a unique protocol which permits it to provide a *uniform interface*, like in the famous and widely applied example REpresentational State Transfer (REST) and RESTful Web Services which uses HTTP and its vocabulary (POST, GET, PUT, DELETE respectively for CRUD) for requests. Another weakness of SOA is that what matters in SOA seem to be the functionality *supposed* to be provided by hidden resources without ever worrying about what really *can* be provided by the resource at the moment of the request due to its internal semantics and constraints. ROA deals with this issue by providing a *state* of the resource as response to a request with all the information and functions that *can* be accessed at the moment. In [Del14] a new architectural style is proposed by combining REST and SOA to get the advantages of both to improve interoperability and facilitate the implementation in IoT.

In this thesis, our proposition will adopt the ROA approach in different abstraction and service level to conserve the flexibility coming from the loose coupling and reusability of SOA while respecting the internal semantics and eventual temporal constraints of each resource entity.

2.3.4.2 An open source platform: OpenHAB

OpenHAB (open Home Automation Bus)[16] is an open source platform running on top of OGSi Equinox, the same as the previously introduced OM2M in 2.2.2, which makes it modular and extensible in functionalities. Its goal is to integrate home automation systems and technologies in a single solution so that they can communicate and be managed via a uniform interface. It uses the notion of *item*, a data centric functional atomic building block, which represents a source of data: if an appliance provides two sources of data, say temperature and energy consumption information, it is modeled by 2 items. These *items* are the ones users see on the user interface and implied in the control rules in form of script.

OpenHAB connects devices or sub-systems using different technologies by *bindings* which are OSGi plugins running on top of the core and other basic modules of Open HAB and added/removed on runtime according to the need. Today, the available bindings include most of the popular protocols, such as KNX, Z-Wave, Insteon, and smart devices, such as Koubachi (plant care) and Netatmo (weather station). Items can be related to specific user-defined groups so that they can be monitored, displayed and controlled as an integrated body. OpenHAB allows user to define their own control rules in form of a script concerning the already connected *items*. It uses Drools open source Java rule engine[17] where rules are described in the format of Condition-Action.

The fact that OpenHAB is a java based platform makes it run on any operating system with a java virtual machine, including embedded platforms such as Raspberry Pi , BeagleBone Black. It also provides multi-platform UI (iOS, Android, Web browser) to monitor and control home devices. OpenHAB joint the Eclipse SmartHome project for better support and further development.

[16] http://www.openhab.org
[17] http://www.drools.org/

OpenHAB provides a platform that puts together the fragmented Smart Home market by establishing a common interface to access all. It is flexible, modular and lightweight that can run on embedded systems. It provides also user-friendly UI and programming support for end users to see the state of the system and to define their control rules easily. However, its original goal makes it focus on connecting smart objects to provide interfaces based on unified device abstraction (here the term "device" means also the individual sensors and actuators). Service, in the sense of generic functions provided to applications, and self-configuration are not in its scope. Objects should be declared explicitly to be connected and groups should be specified manually. The notion of *item* may also create confusion because it is not equivalent to an object, which is the opposite to intuition.

2.4 Semantic modeling

The term *Ontology* takes its origin in philosophy which studies the nature of being, existing and their categories and relations. It is used in computer science and information science to represent a knowledge about the world. It is defined as a formal specification of the concepts and relationships of terms and consists often of taxonomic hierarchies to categorize terms, class definitions, subsumption relations [Gru93]. Moreover, it provides *a priority* knowledge, most coming from the real world, defining the semantics over the categorization and the relations, which makes it differ from traditional information classification or organization which may be arbitrary. In the scope of this thesis, the term *ontology* is used in the sense in computer science or information science.

2.4.1 Semantic approach for IoT and Smart Environment

In our target scope of application domain, we should expect the number of entities to be managed extremely high that the issues in maintaining the interoperability of these entities becomes more and more challenging, such as how to present, store, interconnect, organize information generated by the entities. To deal with these problems, semantic approaches could play a key role. Semantic technologies aim to provide a *globally understanding* between "things" to enable the self-capabilities of the target context, for instance self-discovery, auto-configuration, context awareness in the domain of IoT. Semantic solutions can reduce greatly the amount of work to make the devices implementing the current existing various standard inter operate and do not require any extra work from the manufacture. The importance of semantics to the research and development of IoT from different aspects including interoperability, resource discovery, reasoning and interpretation, has been shown and discussed in [Bar+12].

Some experimental platforms for IoT and Smart Environments have been developed in the recent years. SESAME-S aims to enhance energy optimization and efficiency in Smart Home and Office using semantical technologies [Fen+13]. Together with energy efficiency applications, the platform hosts ontologies, rules and services to create a flexible and user-friendly system, and has already been installing in real building in the trail phase. In [SCM10], the authors propose a solution of *semanticization* of multiple existing standards to make them interoperate. The main idea is to extract the semantics provided by

the specifications of the standard and wrap them into a universally understood semantic language, which makes heterogeneous devices work together without modifying the native standards.

2.4.2 Domain ontologies

Multiple definitions of the term *ontology* exist in the literature with some slight differences [Gru93]; [NM+01]; [Gua98]. To be simple, we take the basic and common parts of these definitions for the purpose of this thesis: an ontology describes a hierarchy of concepts (**classes** or **categories**) related by subsumption relationships or other relationships, which may be described as a triple $conceptA - relationship - conceptB$. **Properties** may be added to each concept to describe their features and attributes as well as restrictions. An ontology is an abstract model of the reality, together with the individual **instances** of classes, establishing a **knowledge base** over the modeling target domain.

By contrast to an upper or foundation ontology which tries to describe the very general concepts with a widely understood common vocabulary across multiple domains, a *domain ontology* or *domain-specific ontology* only tries to model one domain of interest with particular vocabulary and semantics, for example, an ontology for the sensor network, an ontology for Smart Home that we will present in the following part of this section. A domain ontology can be considered as an *under ontology* with more details a finer-grained version of one part of an upper ontology which provides a semantic interoperability over its under ontologies. Domain specific ontology is more practical to use as it provides more detailed and useful information/knowledge about the domain, whereas an top-level ontology is often domain- and problem-independent that may not suit for a particular application.

In the target scope of the thesis, a semantic solution is often dependent of a domain specific ontology, as presented in the examples in 2.4.1. Apart of those coming with a systematic solution, a lot of research work has been dedicated to proposition and establishment of functional domain ontologies that can be used as an input resource to a system. The following are just some examples among many propositions.

Sensor network ontology. Sensor networks are essential and enabling technologies for IoT and Smart Environment. Data collected by a sensor network are direct information about the context and things in the context. The *Semantic Sensor Network Ontology* (SSN ontology)[18] is developed by the W3C Semantic Sensor Networks Incubator Group (SSN-XG) and has been admitted since several years in the domain. It can describe sensors, sensing, the measurement capabilities of sensors, the observations that result from sensing, and deployments in which sensors are used. These modules are all inter-connected in the ontology. Sensors are not limited to physical sensing devices: any device or a computational process that can estimate or calculate the value of a phenomenon can play the role of sensor. As sensors are often constrained by its memory, computation capability, power availability, semantics which is basis of inference or reasoning techniques is needed to formally represent these constraints. That's why SSN ontology also aims to extend the Sensor Model Language

[18] http://www.w3.org/2005/Incubator/ssn/ssnx/ssn

(SensorML), one of the four SWE languages, to support semantic annotations to improve semantic interoperability. Another approach to establish a sensor ontology is to create an ontology that links other domain ontologies. SenMESO ontology (Sensor Measurements Ontology) [Gyr13] is such an ontology which interfaces other ontologies at semantically identical categories in both ontologies to have additional information.

Ontology for Smart Environment. There is very active work on smart environment ontology proposals with various emphasis. Some of them try to cover a sub domain of the smart environment, like *homeActivity*[19] and *homeWeather*[20] in the *Linked Open Vocabularies (LOV)* project[21]; some of them try to accomplish a goal like home energy management [Rei+11]. An survey[GNP13] has shown that ontologies from different horizons can be connected and extended to create a comprehensive knowledge base that covers all the facets of a very rich domain like the Smart Home. We describe in the following 2 wildly cited Smart Environment ontologies which gave us inspiration for the ontology described in Contribution.

- **DogOnt**[BC08] is a house modeling ontology designed to fit real world domotic system capabilities and to support interoperation between currently available and future home automation system solutions. It has several particularities which make it more flexible and more appreciated by the community: (a) it dedicates a branch of hierarchical tree to *controllable objects* including appliances and house plants like HVAC system, which facilitates the implementation of control applications by knowing the *controllability* of addressed objects; (b) it suggests the concept of *state* representing a stable functional state of the modeling target, which provides more information and some restrictions during not only instantiation phase but also runtime. This ontology gives us some inspiration of properties and common entities to take into account for our ontology.
- **BONSAI**[Sta+12] is an ontology for enabling Ambient Intelligence in a Smart Building. It relies and extends existing ontologies in the domain such as CoDAMos and OWL-S in order to create a comprehensive and more operation-able ontology for Smart Building service-oriented system. It has 5 sub-clusters which are linked together to model every aspect needed in a SOA Smart building system: (1) Hardware, like appliance and device, which are able to offer or receive services; (2) Context, like location and other contextual parameters; (3) Functionality, including parameters like power consumption and actions by actuators; (4) Service, imported from OWL-S; (5) QoS, system performance related concept enabling optimization solutions. The target domain of this ontology has intersection with our proposition.

A new reference ontology, **Smart Appliances REFerence (SAREF) ontology**[22], targeting the appliances in the home and building areas is under work. It started from existing semantic assets, tries to separate and recombine them, with the ultimate objective

[19] http://sensormeasurement.appspot.com/ont/home/homeActivity
[20] https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/WeatherOntology.owl
[21] http://lov.okfn.org/dataset/lov/
[22] https://sites.google.com/site/smartappliancesproject/ontologies/reference-ontology

to become a part of ETSI M2M standard as a unified reference ontology. The basic blocks of SAREF are *device*, *function* and *service*. The start point is *device*, which is an object designed to accomplish one or more *functions*, and provide a *service* representing a *function* to a network to make it discoverable, registrable and remotely controllable.

2.4.3 Tools for ontologies

Exploitation of ontology would not be possible without any tools: from standardized format for description to query language for retrieve useful information, from local data structure for managing one or several limited-size ontologies to access technologies and methods to link distributed ontology resources on the internet.

Representation and description. The most popular and adopted language is the **W3C Web Ontology Language (OWL)**, designed to represent rich and complex knowledge about things, groups of things, and relations between things, and readable by computer programs to facilitate machine-to-machine information exchange without human inference. It uses RDF/XML as the primary exchange syntax with vocabularies defined as OWL formal semantics to describe the knowledge to make the knowledge more accessible to automated processes. At the very beginning of the language conception, it is assumed to allow information to be collected from distributed resources, which is, ontologies can be referenced by URI, imported by other ontologies, and complemented and extended. **Resource Description Framework (RDF)** is a W3C standard model for data interchange on the Web. Its core data structure is a set of triples (subject-predicate-object) which can be easily visualized and transformed as a directed labeled graph (called RDF graph) where resources are represented by the graph nodes linked by the edges representing the named link. URI are applied to both resources and links that can be easily accessed on the Web. These characteristics make document in a format of any of the RDF-based languages, including OWL, easily exploitable as a graph and easily reusable over the entire Web as *linked data*. Also, another data model is defined in RDF: RDF dataset, which is a collection of RDF graphs known as *named graphs*. To reach a resource in a named graph, instead of using a triple subject-predicate-object as in a single RDF graph, a quad having the form graphname-subject-predicate-object is more adequate. In our contribution, our ontology consists relationships at different levels. In order to separate them based on their nature, we will consider that these edges belong to separated graphs sharing the same nodes.

Query and inference. Queries are made to retrieve and manipulate useful information from the knowledge base built on relative ontology(ies). Inferences are needed to generate new knowledge from existing one according to pre-defined rules. SPARQL (SPARQL Query Language for RDF) is a W3C Recommendation[23], whose name indicates that it is created for making queries against data stored as native RDF or viewed as RDF via middleware. It has numerous implementations including Jena that we choose for our implementation. SPIN (SPARQLE Inference Notation) defines a systematic framework on

[23] A W3C Recommandation is a mature standard endorsed by W3C that deployment and implementation are encouraged.

how to use SPARQL queries to constraint the values of a class, to generate new triples based on existing RDF graph according to inference rules and to initiate new instances with default values. It takes the object-oriented approach in which embedded SPARQL queries play a similar role to functions and methods. As SPIN is entirely represented in RDF, rules and constraints can be shared on the web together with the class definitions they are associated with.

It should not confuse the *inference rule/constraint* with the *control rule/constraint* that will be used later to express control objectives in the target domain ensured by formal method.

2.5 Data sharing

Data sharing is essential for future research in many fields, including in the present IoT and SE. Data can be reused for multiple objectives, more accurate information can be retrieved from big amount of data, research results can be directly accessed by other researchers. Traditional database such as SQL rational database for data storing and sharing have met the bottleneck due to its nature which makes it not adequate for the very fashionable "big data" and other types of data sharing.

Linked data and persistence support *Linked data* is about to link every piece of data so that person or machine can easily explore the web of data. The idea is to use URI to name every piece of data descried in a standard format (RDF) in which links to other URIs are provided to make a huge web. For the ontologies, as OWL is a RDF-based language, they can be linked and accessed over the web as well as every class, relationship and instance on the basis of linked data.

Graph database A graph database is a database that uses graph structures with nodes, edges, and properties to represent and store data. By definition, a graph database is any storage system that provides index-free adjacency. This means that every element contains a direct pointer to its adjacent element and no index lookups are necessary. Graph databases are based on graph theory that can be used to solve different types of problem, include shortest path calculation, the geodesic calculation (Geodesic Path), concentration measurement (such as intimacy, relationship degree, etc.). a triplestore or a set of triplestores described in RDF can be considered as a specialized graph database. However it has performance problem when the data amount becomes really big and the manipulation on RDF, which is a conceptual data model, depends totally on the chosen implementation. Neo4j[24] is a general graph database implementation in Java that has its property data model (Blueprints) with features that implement results of graph theory mentioned above. Its performance has been tested satisfied in traversing the graph (2000 relationships per millisecond according to its manual). As RDF is a standard and widely used, tools are developed for storing and querying RDF in Neo4j[25].

[24] http://www.neo4j.org/
[25] https://github.com/tinkerpop/blueprints/wiki/Sail-Ouplementation

Our proposition is designed to be able to evolve in the future to receive and support the above data sharing and storage technologies with planned interfaces. For the present time, some more traditional solutions will be used for a first validation of the proposition.

2.6 Applications

Applications in the domain of IoT and SE are numerous that we are not able to list them all individually. Thus, we show in the following some common categories of applications according to their main objective.

2.6.1 Safety and Security

Safety and security are essential for living people in all environment, without which any comfort or efficiency is only empty talk. Sensors deployed in the environment can detect events and significant parameters continuously. Thanks to these raw data, the intelligence in the environment could determine the potential or present danger according to a single event or a temporal coincidence of several events and parameters which avoids inappropriate false alarm. Following the revealed potential or present danger, the system reacts according to the given plan by actuating the relevant pieces of equipment via actuators. Some typical example applications are: home/building anti-intrusion system responding to detection of breaking in; fire prevention system reacting to detection of smoke or an over-heating due to dysfunction of a piece of equipment; auto-lighting system switching on a lamp when the presence of some one is detected in the corridor. These examples are taken from the home/building area, however, they are transposable to city scale. For instance, light control in a corridor is easily applicable to a street, over-heating detection at one node of the home/building scale system is similar to detection of an over-consumption at one location in a block to prevent a black-out at a larger scale.

We can notice that these applications have a strong temporal constraint: the system should react quickly enough to stop any damage, which makes the popular "cloud-based" solutions not appropriate because of the uncontrolled delay. Moreover, they should respond to a minimum level of reliability to ensure the effectiveness of such critical functionality that the "best-effort" approach is not adequate.

2.6.2 Energy Management

Like safety and security management, energy management applies at different scales in the domain of IoT and SE: from the highest level of the global (smart) grid to the lower level of micro-grid covering a neighborhood with the capability of energy generation (i.e. solar photovoltaics), down to the smallest scale of individual building or home. The benefit of the applications in this category is intuitive and obvious because they reduce the energy bill, which makes a great number of such applications or equipment already available on the market. These applications can be very complex taking into account of multiple sources of information, for example, an adjustment of a heating system power depending on the time

in the day, inside and outside temperature, presence and activity detected and current and future meteorological information. Also, they can be every basic prevention of waste of energy, for example, switch light off when it is not needed.

2.6.3 Comfort enhancement

The objective of the applications in this category is to provide a better living experience for inhabitants. They adjust the environment parameters, like sound, light, temperature, in accordance with the ongoing activity. They can take over repetitive or tedious tasks using automation logic in order to save people's time. The detection of ongoing activity is crucial for this category because the definition of "comfort" would be opposite in different situations. For example, the light should be adjusted to bright when someone is working to have a better productivity whereas it should be faint when someone is watching TV or a film. Sometimes the applications in this category can also have opposite objective than the ones in energy management category, for example, the comfort enhancement application aims a higher temperature while the other aims a lower temperature to save energy. There is no absolute manner to manage this kind of conflicts that the choice should be left to the end user to make the compromise.

2.6.4 Health and assistance

Health and assistance applications were first developed to improve the life quality of elder and ill people. These people are vulnerable and often lack of companion. Health and assistance applications can track their daily activities to verify if they respect the routine, i.e. taking medications, eat properly, doing exercises. Moreover, their health conditions can be sensed automatically by available devices or raised explicitly by the monitored person to be sent to the doctor, which enables telemedicine solutions, such as remote medication prescription. For normal people, benefits of these applications are numerous as they facilitate the monitoring of daily body conditions and the tracking of daily activities to encourage them to keep healthy. Many solutions are available on the market, including numerous connected devices, such as connected bracelet/watch, smart balance, etc.

2.7 Summary and discussion

In this chapter, we first introduced the common characteristics of our target scope based on which is built our contributions: openness, dynamicity, heterogeneity, similarity. These characteristics bring challenges and requirements for research and development of infrastructures and applications in this specific domain.

Facing the above properties and challenges, especially openness and heterogeneity, there are numerous on-going activities in standardization. Some of them define the general architecture and requirements for the entire domain in a high level abstraction, like the ITU-T reference model; others are more specific in application domain such as ETSI M2M for M2M, HGi for Smart Home, or in technology such as DNS for resource identification.

While the standards design the outline of the development in the target domain, enabling technologies such as WSAN and identification technologies build the base and provide the guidelines following which we are able to make our proposition in the thesis.

In order to deal with the dynamicity problem and the quantity of entities to be managed in the domain, the concept of ontology is used to enable and facilitate the self-identification, self-organization and self-management among potentially numerous entities. Some work on the semantics and ontology research in IoT/SE domain, such as ontologies for sensor network (SSN) and those for Smart Home/Building (DogOnt, BONSAI) are cited which have inspired us to design our ontology and will be used as external ontologies in our proposition, as well as some common practice and tools for ontology design, implementation and usage.

We have also noticed through this chapter that IoT and SE is such an immense research and application field where problems are numerous, everything is still evolving and changing though some standards or references are present. Research activities are dynamic that various propositions exist for one problem in different circumstances. Disciplines meet here to provide new idea and new capabilities for this emerging field. Concerning our proposition, it is clear that some existing approaches cited in this chapter are not adopted even their results have been proved, such as EPC-RFID framework, and that some work will be taken as a priori knowledge even there are a lot of ongoing work going much deeper, such as sensor data fusion, ontology. Our proposition and contribution will concentrate on the control aspect of the target scope treated until today as a "naive" problem which is in reality *not*. In the next chapter, we will discuss this aspect more in detail.

Chapter 3

Models and Discrete Control

Contents

3.1 Common modeling approaches in IoT/SE

The target domain has a great intersection with the so-called *Cyber-Physical System* (CPS) which integrates computation and physical processes. Its potentials have been recognized recent years attracting a lot of investments. Challenges of CPS modeling have been identified and studied from which the IoT/SE can benefits a lot for its own modeling problems. Numerous models and modeling methods have been applied to cope with different objectives in IoT/SE either by adopting a software engineering oriented approach or an embedded system oriented one. No method dominates the others as long as it provides a appropriate level of abstraction, understandability and accuracy to the given problem [Sel03].

3.1.1 Object-oriented modeling

Object-oriented modeling (OOM) is a widely applied approach in software engineering to model applications, services and systems using object-oriented paradigm throughout the

entire development life cycles. The Object-oriented programming paradigm consists of using the concept of *object* which is an abstract data type with data fields (attributes) and codes (procedures or methods), as well as some fundamental principles: *class inheritance* and *encapsulation*. A *class* is a user-defined data type used as template to *instantiate* an *object*. *Encapsulation* is the feature that each object, an instance of a class, has its own data and is responsible for its own behavior. Two objects of the same class do not share their encapsulated data or methods and any change to one object does not affect the other. *Class inheritance* allows code reuse by making one class "dependent" on an other one using the same implementation or extending it. This *inheritance* only consists of implementation and interface inheritance, but not of the behavioral inheritance or behavior conformity, that no guarantee is provided in objected-oriented philosophy that an object of class B inheriting class A would produce the same result as an object of class A[CHC89]. This modeling approach is supported by modeling languages such as Unified Modeling Language (UML). It is also valuable in realizing finite state machine that we will present in more details in 3.2.1.2.

3.1.2 Agent-based modeling

Agent-based modeling (ABM), also known as *individual-based modeling*, can be understood as a "bottom-up" decentralized and individual-centric approach paying attention to the interactions between individuals. This modeling approach has emerged facing the higher and higher system complexity that makes the design too difficult. At meanwhile, decentralization has been more and more often as infrastructure like *micro grid* equipped with its own generators in the general electricity grid, as well as in social networks and systems since we pass from the era of web 1.0 to web 2.0 where everyone contributes the content to be consumed by everyone. ABM focuses directly on the modeling of individual entities, called *agents*, their behaviors and possible interactions between them and the context. They are then put them into an environment and the simulation is run with assumption that complex global system behavior can be built from local behavior interactions. Application domains are various such as biology, ecology, social science. By definition, the *agents* in ABM refers to autonomous decision-making units with diverse characteristics[Cas06]. This diversity of individuals and the focus on "bottom-up" interactions make perfect analogy to the IoT and Smart Environments whose characteristics have been identified in 2.1, which makes this modeling approach widely applied and developed in this domain such as [Bat07][For+13][Per+12].

JADE (JAVA Agent DEvelopment Framework) is an open-source framework to facilitate the implementation and deployment of multi-agent applications. It provides an implementation of agent model quite "primitive" implementing only the very essential properties such as autonomy and inter-agent messaging mechanism which are easily to be specified in a given implementation. It offers a communication architecture based on the peer-to-peer communication model using the state-of-the-art distributed object technology embedded in Java Runtime. Besides, it offers graphical tools for debugging and deploying. Its scalability has been proven with respect to the number of agents and the number of simultaneous conversations for a single agent [Cor+02]. This platform has been used and maintained for over 10 years, and extended to other active platforms such as WADE

(Workflows and Agents Development Environment) and AMUSE (Agent-based Multi-User Social Environment).

Weakness and requirements. Having a closer look to the two above *mainstream* informational modeling approaches, we notice that they have been more or less developed within *transformational* systems where design methods are naive relying on unquestioned assumptions compared to modeling methods practiced in reactive systems. Though the applications in our target domain do not present as much constraint as in reactive systems, formal method from reactive systems integrating a temporal semantics of modeling targets is more than welcome in order to overcome the weakness coming from the nature of transformational systems.

3.2 Automaton-based modeling

The proposed modeling framework for IoT/SE is based on automaton after the identification of weakness of classic modeling approaches in the previous paragraph. The choice of this model is generally because the IoT/SE system can be conveniently modeled as an DES and automaton is a formal description of DES behavior. In this specific domain, a lot of information is discrete such as the command by a human or an automatic rule, pieces of data sensed by sensors at a given frequency, and the states of physical entities are discrete like on/off of an appliance by nature. Discrete information can be considered as discrete events occurring at time instances instead of overtime which will cause changes of discrete states of the systems.

In this section, we begin with the definition of automata that we use in the present thesis, then we introduce some common operations and extensions on automata, finally we describe the language that we use for the rest of the thesis. Note that in the thesis, the term *automaton* is used interchangeably with *state machine*. If it is not specifically indicated, *automaton* and *state machine* mean *finite* automaton and state machine, which is often referred by the abbreviation FSM (Finite State Machine).

3.2.1 Definition of Automata

There are many variations of automata such as deterministic/nondeterministic on transition, finite/infinite on state. Automaton is also often considered as a representation of formal language according to well-defined rules, thus definitions of automata can take a formal language approach. In this thesis, we consider only the *deterministic finite state automaton* and adopt the definition in [LS11] from a *discrete dynamics* approach which is a machine reacting to input and producing output.

A finite state machine is a 5-tuple

$$S =< \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} >$$

where:

- $\mathcal{Q}$ is a finite set of states;
- $q_0 \in \mathcal{Q}$ is the initial state of the S;
- $\mathcal{I}$ is a finite set of input events;
- $\mathcal{O}$ is a finite set of output events;
- $\mathcal{T}$ is transition relation that is a subset of $\mathcal{Q} \times Bool(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$, such that $Bool(\mathcal{I})$ is the set of Boolean expressions of $\mathcal{I}$ and $\mathcal{O}^*$ is the power set of $\mathcal{O}$.

Transitions between states govern the discrete dynamics of the FSM. Each transition, denoted by $q \xrightarrow{g/a} q'$, is labeled by g/a, where guard $g \in Bool(\mathcal{I})$ should be evaluated *true* when the transition should be taken, and action $a \in \mathcal{O}^*$ is a conjunction of output events, emitted when the transition is taken.

3.2.1.1 Hierarchical automata

Hierarchical automata consist of FSMs whose states are themselves an FSM and semantics should be defined specifically to describe how the *component* FSM reacts regarding to the *superstate* which nest it. The advantage of a hierarchical state machine instead of using its equivalent flattened version [Yan00] is that it is a more compact notation so that it is more "human-readable" especially when the number of states grows. It is also very convenient to model something hierarchical or modular *by nature* such as the example of a digital watch which has several functional modes given in [Mar91].

We will not detail any specific semantics of hierarchical automata here because there are various and the choice should be made in accordance with what is being modeled. In 3.2.2, we will describe the semantics of hierarchy of the given languages.

3.2.1.2 StateCharts and UML state machine

StateCharts [Har87] first emerged in 1983 to specifying the behavior of an avionics system which is a typical example of reactive system [Har07]. It provides a visual tool (diagrams, graphs) based on rigorous and precise mathematical meanings for the fundamental concepts in a reactive system such as a state, a transition. It supports modular and hierarchical description of system behavior and orthogonality between components. These two operations can be mixed at any level. However, it was still a specification language for real-time embedded systems without a precise semantics until the StateMate [Har+90] which implements a specific semantic adopting the synchrony hypothesis (see: 3.2.2).

StateCharts has object-oriented variants among which the most famous example is the UML state machine. The main idea of OO variant of StateCharts is to use *class diagram* to structure the object classes and each class is associated to a statechart which describes precisely its behavior based on a rigorous semantic definition. The semantics in these OO variations are not completely the same: StateCharts, implemented by StateMate is not OO and is synchronous, while OO variants are basically asynchronous as the transitions

are executed in a run-to-completion manner where the events are queued up to be handled to the system when a previous event has been completely treated [HG96].

StateCharts is well tool-supported that the models can be directly translated into executable codes like C, C++, Java. It is also the origin of some automaton-based synchronous languages such as Argos [MR01].

3.2.1.3 Synchronous composition of automata in parallel

Synchronous composition of automata in parallel is one pattern of concurrent composition of state machines and is denoted by $\|$.The result of a synchronous parallel composition of automata is itself an automaton and this operation conserves the determinism held by component automata (it means the result is deterministic if the components are deterministic). Given 2 FSM $S_i =< \mathcal{Q}_i, q_{i0}, \mathcal{I}_i, \mathcal{O}_i, \mathcal{T}_i >, i = 1, 2$, with $\mathcal{Q}_1 \bigcap \mathcal{Q}_2 = \emptyset$. Their synchronous composition in parallel is $S_1 \parallel S_2 =< \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{10}, q_{20}), \mathcal{I}_1 \times \mathcal{I}_2, \mathcal{O}_1 \times \mathcal{O}_2, \mathcal{T} >$, where $\mathcal{T} = \{(q_1, q_2) \xrightarrow{g_1 \wedge g_2 / a_1 \wedge a_2} (q_1', q_2') | q_1 \xrightarrow{g_1/a_1} q_1' \in \mathcal{T}_1, q_2 \xrightarrow{g_2/a_2} q_2' \in \mathcal{T}_2\}$. Composed state (q_1, q_2) is called a *macro state*, where q_1 and q_2 are its *component states*. Figure 3.1 shows an example of synchronous composition.

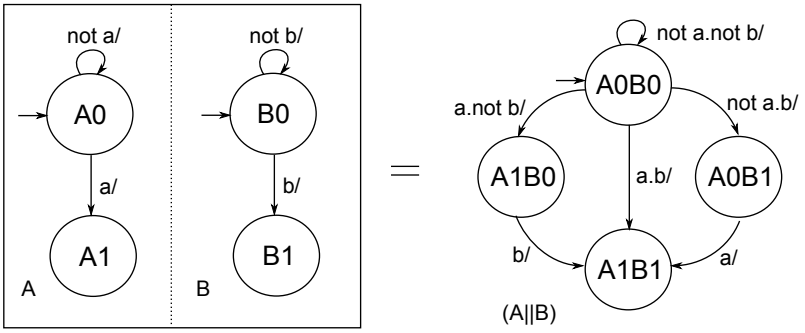


Figure 3.1 – Synchronous composition example

3.2.2 Synchronous programming

The synchronous approach dates from the mid 1980s and was first proposed to provide a rigorous mathematical semantics for *reactive* systems, distinct from *transformational* systems which take input data, perform computation and produce output data. It models reactive systems by *synchronous-reactive* model which is a discrete system where signals occur at ticks of a global clock, and each reaction to such signals is considered simultaneous and instantaneous. The output of the system is assumed as simultaneous with their input, called the *synchrony hypothesis*. Real executions do not literally occur simultaneously nor instantaneously, and outputs are not really simultaneous with the inputs, but a correct system behavior model must behave as if the output has been computed to reach a new global system state before the next clock tick. A number of synchronous languages have been developed since based on the hypothesis, such as Esterel [BG92], Lustre [Hal+91], Signal [LeG+91], Agos. Note that synchronous languages are high level *programming* languages whose programming environments provide compilers towards target software

or hardware environments, as well as validation tools as these languages rely on formal semantics. StateCharts has also many features of synchronous languages but it is for model specifications. The automaton-based synchronous languages (e.g. Agos, Heptagon) will be described in the next subsection.

3.2.2.1 Automaton-based synchronous languages

Argos[MR01] is a synchronous language allowing to combine Boolean Mealy machine in parallel and hierarchical way. It takes its origin in StateCharts with a very similar graphical syntax, however, it does not support the *multi-level arrows* which represent the transitions between two states in different hierarchical levels. It solves *causality* (similar to deadlock) and *modularity* issues in StateCharts based languages. It has verification tools and auto code generation compiler which produces data-flow equations under a format which can be compiled into C.

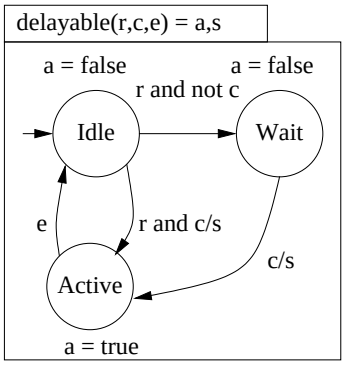
An other example is the Heptagon synchronous language inspired by Lucid Synchrone [CPP05] and mode automata which aims to help building system to declare clear "running modes" [MR03]. It is the programming language we use in this thesis to describe the FSMs for the discrete control synthesis to perform with. Thus, we will introduce it with more details in the next subsection. Some data-flow languages provide also automaton-based formalism to deal with control-oriented designs, such as Synchart [Cha96], Esterel [BG92].

3.2.2.2 Heptagon language

Heptagon language programs behave as synchronous automata, with both parallel and hierarchical composition. For scalability and abstraction purpose, the synchronous programs are considered structured in *nodes*, which has a name, a set of input and output flows, and equations defining outputs as functions of inputs describing the component *behavior*. The node type considered here is used to encode mode automata which is a mix of dataflow equations and more imperative automaton-based programming. Each state of an automaton is associated with such a node with equations or a mode automaton. Its basic behavior is that at each reaction step, according to inputs and current state values, equations associated to the current state produce outputs, and conditions on transitions are evaluated in order to determine the state for the next step.

An example of Heptagon node is a delayable task control, for which figure 3.2 gives a graphical and a textual syntax. The node's name is `delayable`, with 3 input flows `r`, `c`, `e` and 2 output flows `a`, `s`. It has 3 states: `Idle`, `Wait` and `Active`. At the initial state `Idle`, transition can be taken upon the condition given on the inputs: if `r` and `c` are `true` then it goes to state `Active`, until `e` becomes true to get it back to `Idle`; if `r` is `true` and `c` is `false` which blocks the request, it goes to Wait until c is true to be `Active`. The outputs are defined by equations: `a` (active) is defined by different equation in each state and `s` (starting) is `true` when transition towards `Active` is taken.

The nodes can be reused by instantiation, and composed in parallel or in a hierarchical




```
node delayable(r,c,e:bool)
returns(a,s:bool)
let
  automaton
    state Idle do
      a=false; s=r and c;
    until r and c then Active
    | r and not c then Wait
    state Wait do
      a=false; s=c;
    until c then Active
    state Active do
      a=true; s=false;
    until e then Idle
  end
tel
```

Figure 3.2 – The graphical and textual syntax of a delayable task

```
node twotasks (r1,c1,e1,r2,c2,e2:bool)
returns (a1,1s,a2,s2: bool)
let
  (a1,s1) = delayable(r1,c1,e1);
  (a2,s2) = delayable(r2,c2,e2);
tel
```

Figure 3.3 – The textual syntax of two delayable tasks in parallel

way. In the parallel composition, the global behavior is defined as a step by having component nodes taking a step simultaneously. An example of parallel automaton is shown in figure 3.3: a global node `twotasks` with 2 `delayable` node instances put in parallel (noted by ";"). In the case of hierarchical composition, a node instance is inside a state and defined the behavior of the state when it is active.

The Heptagon compilation produces executable code such as C or Java. The generated code has 2 main functions: `step` and `reset`. `reset` initializes the state of the program and `step` executes one reaction. It takes input values, computes the next state on internal variables and returns output values of current state of the program.

3.3 Supervisory control and Discrete Controller Synthesis (DCS)

For a DES whose behavior must be modified by feedback control to respect a given set of specifications, a feedback control loop needs to be designed with a *supervisor*, also called *supervisory controller*, whose goal is to alter the behavior of the *uncontrolled* DES to make

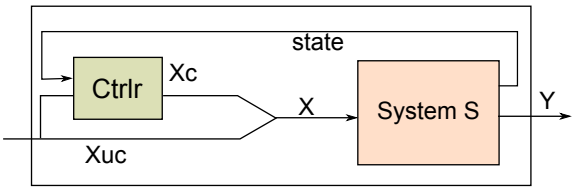


Figure 3.4 – Controlled system with synthesized discrete controller in a closed-loop

it satisfactory. DCS is a formal operation on DES to find such a supervisor. The advantage of this approach is that it separates the concept of open loop dynamics (here the DES) from feedback control , which allows the autonomic analysis and control of DES w.r.t. a given specification of control objectives.

3.3.1 The Ramadge and Wonham Framework

Discrete Controller Synthesis was first introduced in the 80's in [RW87] to deal with control and coordination problems of DES presented as e.g. a set of automata (originally uncontrolled). It defines constructive methods that ensure, a priori, required properties on a system behavior: it does not only verify if a supervisor exists to enforce the property, it constructs one if it exists. From a such a DES and a set of properties to be satisfied, the synthesis produces the constrained system so that only behaviors respecting required properties are authorized. The principle of DCS is as follows. The inputs X of the DES are partitioned into two subsets: $X = X_c \cup X_{uc}$, where X_c is the set of *controllable* events which can be disabled by the controller, and X_{uc} is the set of *uncontrollable* events which cannot be prevented from happening. It is applied with a given control objective: a property that has to be enforced by control, which is expressed in terms of the system's output Y. A controller $Ctrlr$ is calculated by DCS algorithm which consists of exploration of the complete state space, the constraint (also called *control pattern*) controllable variables, depending on current state, for any value of uncontrollable ones, so that remaining behaviors satisfy the objective. It is then composed with the original system S, taking X_{uc} and the current state of S[1] as input to produce the values of X_c which are enforcing the control objective.

There can be several controllers that meet the same control objective. In the extreme case, a controller can inhibit any state transition in order to avoid the invalid states, which is apparently not interesting for the target system. The controller resulted from the DCS process is called *maximally permissive* which means it inhibits the minimum possible behaviors to respect the constraints to ensure a largest set of correct behaviors of the originally uncontrolled system.

DCS procedure is automatic and implemented in the tool Sigali [Mar+00].

[1]The current state of S is not equivalent to the output values of S. For graphical clarity and pedagogical reason, we consider here that the output values contain both the information of current state and the "real" output Y.

3.3.2 Control objectives

In the thesis, we will consider logical control objectives, defined in terms of states and transitions of a discrete event system modeled as automata. The synthesis algorithms corresponding to these objectives exist in the literature and have been implemented in Sigali.

The following two logical control objectives are considered:

- invariance of a subset of states E. A function $S' = make\_invariant(S, E)$ that synthesizes and returns a controllable system S' such that the controllable transitions leading to states $q_{i+1} \notin E$ are inhibited, as well as those leading to states from where a sequence of uncontrollable transitions can lead to such states $q_{i+1} \notin E$.

- reachability of a subset of states E. A function $S' = keep\_reachable(S, E)$ that synthesizes and returns a controllable system S' such that the controllable transitions entering subsets of states from where E is not reachable are disabled. Note that making E invariant is equivalent to making states not in E unreachable.

3.3.3 BZR synchronous programming language

BZR[2] is a high level programming language extending the Heptagon language with a new behavior *contract* which enables the separation of concerns between description of system to be managed (in Heptagon nodes) and the control objectives (expressed as *contract*). The language is a mixed of imperative and declarative styles: the system behavior is described in imperative automaton-based programming and the contract specifying control objectives is enforced in a declarative way. It encapsulates DCS in its compilation process and constructs a controller following the compilation if it exists, which makes DCS accessible to those who do not know the its detail techniques. The synthesized controller produced as a BZR program is then compiled towards sequential code (C, Java) as which has already been done on the synchronous imperative part (automata and equations) of the initial BZR program.

Figure 3.5 shows concretely an example of a BZR program with a contract coordinating two instances of the delayable node presented in figure 3.2. The `twotasks` node has a `with` part that declares controllable variables c_1 and c_2, and the `enforce` part that asserts the property to be enforced by DCS. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time: `not (a1 and a2)`. Thus, c_1 and c_2 will be used by the computed controller to block some requests, leading automata of tasks to the waiting state whenever the other task is active. The constraint produced by DCS may have several solutions: the BZR compiler generates deterministic executable code by giving priority, for each controllable variable, to value `true` over `false`, and between them, by following the order of declaration in the `with` statement (in the present example c_1 over c_2).

[2] http://bzr.inria.fr

$$
\begin{array}{|l}
\hline
\texttt{twotasks}(r_1, e_1, r_2, e_2) \\
\qquad = a_1, s_1, a_2, s_2 \\
\hline
\texttt{enforce not } (a_1 \texttt{ and } a_2) \\
\hline
\texttt{with } c_1, c_2 \\
\hline
\end{array}
$$

$$
\begin{array}{l}
(a_1, s_1) = \texttt{delayable}(r_1, c_1, e_1)\ ; \\
(a_2, s_2) = \texttt{delayable}(r_2, c_2, e_2)
\end{array}
$$

Figure 3.5 – Delayable tasks: exclusion contract.

3.3.4 Discrete control applications

Control over DES with feedback loop using a synchronous approach is traditionally applied in real-time embedded systems, in safety-critical industries like for nuclear power plants and aircraft control software, or manufacturing systems. The discrete controller synthesis approach has been recently applied in more "ordinary" computing systems concerning component-based modeling [DR10], coordination of autonomic administration loops [Del+14], reconfiguration of adaptive systems implemented on FPGA [An+13], etc. It is still emerging in the emerging IoT/SE domain.

In [GBB13], the authors propose a solution of smart home using the modeling and verification methods of DCS in order to improve the safety in home for disabled people. The main idea of this proposition is to correctly model the system with both "working normally" and "failure" modes and if the system has a problem observed by an observer (in "failure" mode), some behavior should be ensured: for example at least one bulb is on to prevent the "black out" in home. To achieve that, compared to a simple model of bulb with basic OFF and ON states, a new state "fail" is added to keep track on the failure (figure 3.6). This state is uncontrollable, which means the entering and leaving of the state are all triggered by uncontrollable inputs of the system. Thus the control of the system is able to be applied only on transitions between ON and OFF where a controllable variable is placed in order to respect the constraint that at least one bulb is on when there is a problem assuming that two bulbs would never fail on the same time. Such solution ensures a correct automatic adaptation facing a given smart home environment with given characteristics and constraints. However, as pointed out also by the authors, to be able to use this approach, a knowledge of the synchronous framework, DCS, etc. is required which should not be expected from system designers. Also, this solution needs a case-by-case design which is costly.

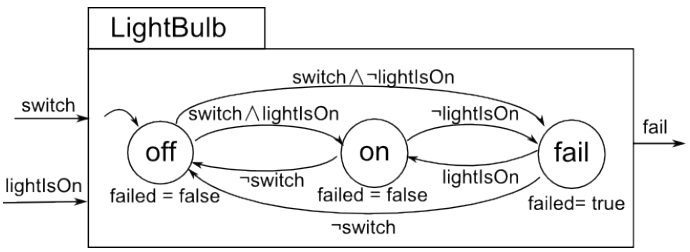


Figure 3.6 – Bulb model with failure

3.4 Related control approaches in IoT/SE

3.4.1 Languages for rule statement and Rule engines

Event-Condition-Action (ECA) rules are defined as a set of active rules which have the structure in the form of: ON Event IF Condition DO Action. When an Event occurs triggering the invocation of the rule, Conditions such as system runtime variable values are verified to decide the execution of Action such as variable value update. It is a widely used language for the high level specification of controllers in adaptive systems, such as CPS and Smart Environments, where devices equipped with sensors and actuators are controlled according to a set of rules. ECA rules are derived from practice, and not constructed from a formal definition in the beginning. Another simplified form of ECA rules are the "if-then" rules which can be considered as the Condition part is always true.

ECA rules are used in many rule engines, especially in those implementing the Rete algorithm [For82] and its variants. An open source example is the Drools[3]. The Rete algorithm consists of traveling a DAG where each node represents a condition or a matching to be verified with entities in the working memory. Rules are executed in the runtime following a "sequential" principle which is to execute the rules in some order (arbitrary from a declarative language or prescribed from an imperative language) commonly adopted in many commercial business rule engines.

An ECA rule-based system is the system whose behavior is controlled by ECA rules. In [CDR14] coordination problems of rules from different origins have been pointed out and an approach to solve these problems has been proposed by using formal methods such as model checking and DCS.

3.5 Summary and discussions

Model-based design has become popular in IoT/SE domain. In this chapter, we first had a closer look at the most popular modeling approaches in IoT/SE: OOM or ABM which present some limits by their nature such as do not implement entity's internal temporal constraints or have no formal verification. Then we presented the approach we use in the proposition: automaton-based modeling often applied in reactive system. Automaton, implicitly finite state in the scope of this thesis, also called FSM, is suitable for system modeling for the target scope as explained in 3.2. In order to obtain a supervisory controller by automatic and formal means, the DCS technique is adopted which is applied on DES whose behavior is described in synchronous language. The BZR synchronous language extending the Heptagon language to describe system behavior encapsulates the DCS in its compilation that produces a restricted system (uncontrolled system + controller) respecting the properties inscribed in the contract.

Based on the models and methods presented in this chapter, and the technologies and semantic tools in the previous chapter, the following chapters will present our contributions

[3] http://drools.jboss.org/

over the control aspect in IoT/SE domain.

Part II

Contributions

Chapter 4

Framework for a shared infrastructure for IoT data abstraction

Contents

In our target scope, we have already recognized their main shared characteristics, including heterogeneity and dynamicity, which implies that they should firstly be organized and managed in an efficient way to be able to hide and reduce the complexity for control service and application. In Chapter 2 we have seen some methods practiced in the domain for this purpose such as horizontalized infrastructure, self-configuration, ontology, based on which we propose our shared infrastructure to enable and support the application of formal control method for generic control objectives.

4.1 Modeling framework overview

In Figure 4.1 we describe a general framework to establish the kind of horizontalized shared infrastructure aiming a systematic and generic bi-directional mediation platform between

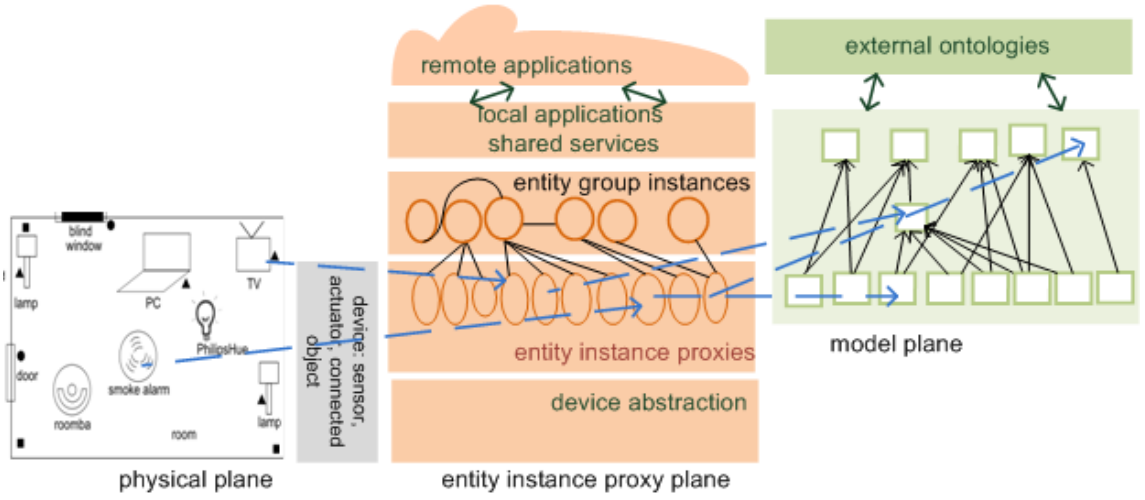


Figure 4.1 – General framework presentation

ICT applications and target physical environments that may be instances of smart homes, buildings or cities, or other IoT environments with similar characteristics.

It is structured conceptually through 3 planes, each representing relationships between physical entities and their informational proxies as a graph in its own dimension. Inter-plane relations are also captured by the infrastructure. These planes correspond to, respectively (figure 4.1, from front to back):

- **Physical plane**. The physical entities of the environment that are represented in the infrastructure and the sensors and actuators used to interface to them.
- **Proxy instance plane**. The instances of software representations (proxies) maintained in the ICT system for these physical entities and devices; external applications interface on this plane.
- **Model plane**. The generic reference models upon which this representation relies; external ontologies and knowledge bases would interface on this plane, providing at least the anchoring of upper categories of this representation.

In this section, most examples illustrating the concept are taken from a home environment instance of one possible configuration among many others. But taking these examples does not exclude the application of the framework to other application domains with similar characteristics, such as building and city. Automata models adopt the notation and style used in the BZR language.

4.2 Physical plane

Entities on this plane have a physical location and an extension in the target physical environment. Also represented in this plane are networked devices that have an effect on their environment through specific sensors and actuators, or separate networked sensors

and actuators . They will also be used, as shown in the following, to provide a surrogate network interface to other entities, including non-networked legacy entities as well as space entities.

4.2.1 Entities/subsystems

In the context of the thesis, entity (short for *physical entity*) refers not only to material *things* like pieces of furniture or equipment, legacy home or office appliances, but also to relevant subsets of space like rooms [Hu14]. The targets of modeling include entities themselves and the subsystems constituted by several of them which are relevant from a user's or an application's perspective. They are assumed without network connection of their own and can be monitored and controlled only through available networked sensors and actuators.

If we give a closer look to our modeling target entities, most of the electrical appliances are equipped with an embedded micro-controller (e.g. washing machine, electrical kettle, micro-oven) which can be considered as well as an autonomous subsystem with control on different components which are microscopical entities. Thus, the border between individual entities and subsystems has become blurred. In the following parts of this thesis, *entity* will denote for both individual entities and subsystems of entities.

4.2.2 ICT devices/connected objects

Nowadays, it is common or even inevitable to have both legacy entities and ICT/connected objects in the same environment. These *smart* objects have themselves the capability of communication with the network, equipped with a universal identity attributed by the manufacturer and network interface based on specific protocols. Examples are the weather station Netatmo[1], the smart bulb Philips Hue[2], connected watches, etc. which are in absolute blooming today. Though they do not need sensor or actuator as intermediary to be integrated into the network, it is possible to model them in the same way as for the legacy entities once they are identified in order to have a more uniform and accurate view of the current target environment and to make more coherent control decision on the global system. The connectivity function part of ICT objects or their embedded sensors and actuators may be considered as an independent part if necessary.

API. Some connected objects are provided with a list of available APIs which enable users or developers to access its features by customized means. For example, we can get the temperature reading of a Netatmo by its API for further use, we can impose a light temperature/brightness to a Hue by giving a value through its API. Not all the objects offer an API. As for the FSM modeling for these entites, the models depend on the available APIs which play the same role as the available sensors and actuators in the modeling of legacy entities. And similar to the principle of legacy entity modeling, not all the available APIs, which correspond to different internal states, are taken into account by the model

[1] https://www.netatmo.com/
[2] http://www2.meethue.com/

as only basic and general states should be captured.

Remark 1. We notice that among the numerous APIs provided with each product, they most adopt a REST style. This is not a coincidence, as we introduced in 2.3.4.1 that REST is very suitable for managing resources whose internal state can evolve without being fully observed or followed by the applications requiring its data. For the same reason, the API towards the outside of the implementation of our proposition will also adopt the REST style. □

4.2.3 Devices (sensors and actuators)

Within the current modeling framework, devices refer to the basic pieces of equipment with network connection which are not the direct target of our proposed modeling[3]. They are typically shared sensors and actuators connected to the ICT network by embedded wireless network interface such as ZigBee, Z-Wave[4], which can play the role of network interface for legacy appliances and entities without one. Thanks to the devices, interactions between the ICT system and entities are made possible indirectly for legacy appliances and other non-networked entities such as rooms. These devices are considered to be stateless[5] transducers and are not themselves the direct target of monitoring and control. They are used as intermediaries to monitor and control other entities. If we take the example of a passive infrared motion sensor, the infrared-radiating objects detected by the sensor, which may be all kinds of complex appliances as well as warm-blooded animals, are the "interesting" target of detection and monitoring whereas the sensor itself is but a dumb transducer. As to RFID technology[6], the thing with an attached RFID tag (e.g. a piece of furniture) is the modeling target, whereas the RFID reader is the networked device which is only used as an intermediary to address it.

Devices can be dedicated to one piece of equipment, but are most often shared, especially for sensors, to provide a consolidated view about the behavior of several "things" (e.g. the home water meter providing clues about all appliances that use water in a home) or about the context (e.g. the ongoing activity in a room or the occupation of a street based information of number of people, light intensity, noise, etc). Some interesting examples of these devices are:

- A sensor in the classical sense which makes measurement on physical quantity and converts it into a signal to be used further. They can be separated ones, e.g. motion/presence detector (based on ultrasound, infrared or other technologies), electric current sensor, water flow sensor, luminosity sensor, or embedded , e.g. the thermostat in a radiator.
- A virtual sensor, extending the classical definition, which represents a useful information provided by the network, e.g. adjusted available power per home during rush hours of electricity consumption provided by the grid.

[3]The distinction between an *entity* and a *device* has been understood and taken up in a similar way by the EU FI-ware (www.fi-ware.eu) and IoT-A (www.iot-a.eu) projects.
[4]Z-Wave is a wireless communications protocol for home automation. http://www.z-wave.com/
[5]*Stateless* means the data transmitted are independent of any historical data
[6]Standard description ISO/IEC 19762

- A separated actuator, e.g. a smart plug[7] attached to the mains cord of an electrical appliance, or an embedded one.

We put explicitly the "device" module between the physical plane and the proxy instance plane in order to underline its position as *interface* between the physical and the information world.

4.3 Model plane

This plane maintains structured knowledge that may be applied to the target environment, taking advantage of the genericity of environments such as homes, buildings or , cities and other environments with similar characteristics that are sufficiently similar for this general knowledge to apply to their different instances. Generic models and their relationships are stored in the plane, eventually interfaced to external ontologies and other knowledge bases able to enrich and complement the current knowledge repository.

Remark 2. The maintained knowledge is not necessarily maintained in the system runtime memory as not all knowledge is required for a given environment instance. It is used as a reference to be consulted by the system runtime which can make the decision of duplicating part of the structure in the memory if needed. It could be considered in someway as a knowledge database often stored in RDF or as a graph database. □

4.3.1 Generic entity finite automata models

Models to be taken into account by the framework are meant to be *as generic as possible* that only some of its most basic and essential properties are captured. FSM is used for these models. Opposite to the technologies attempting to provide an exact and comprehensive model for every existent entity like EPC, UPnP presented in 2.3.1, these "modal" models make it possible to fully identify target entities by the most approximate generic model which may serve as their dynamic proxies for a large set of applications and services.

- **Thing**. Its states are observable and controllable either by the end user, e.g. pushing a button, from the physical world, or by a service or a controller from the ICT system. Control orders are delivered to the entity through either a networked smart plug for mains-connected appliances, or a direct network interface if available. Following are some model examples of home appliances whose input values can be set either by end user or by the system commands :
  - **Lamp**. Its model has 2 states *off* and *on*, also relevant for most electrical and ICT appliances.
  - **Washing machine**. Washing machine has 2 states without power consumption: *off* and *stand by*. The latter allows a temporary pause in the middle of one

[7] A device integrating both an electric current sensor and an actuator which connects to the existing home area power line and allows other equipment to be connected in order to monitor and control them.

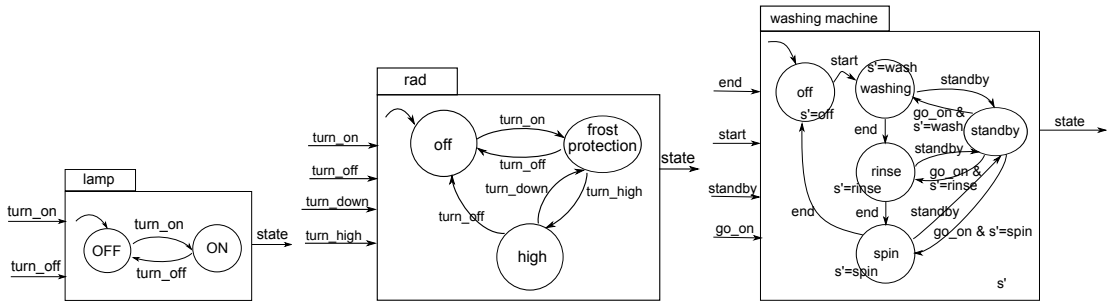


Figure 4.2 – Generic "thing" automaton model: (a)Lamp; (b)Radiator; (c)Washing machine

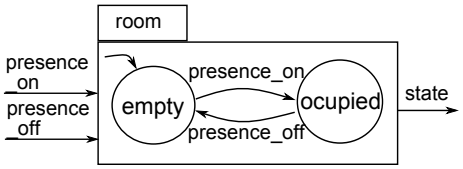


Figure 4.3 – Generic "space" automaton model of room

functional washing phase (upon the reception of a *standby* signal) with the possibility of returning when permitted (*go_ on* signal and global variable *s'* for memory). 3 states *washing*, *rinsing* and *spinning* represent the different steps in a washing cycle existing in all modern washing machines.

– **Radiator**. A radiator has state *off* without power consumption and 2 states *frost protection*, *high* with different set temperature.

- **Subset of space**. Subset of space is usually modeled through states corresponding to different activities, such as *empty*/*occupied* for a room or a street. These states are non-controllable, yet observable through an aggregation and consolidation of data from multiple sensors or external data sources providing useful context information, such as in the example in fig 4.3 the input `presence_on` and `presence_off` which are not a simple value as they appear but are a resulting value from multiple data sources. Another difference is that the subset of space is not directly associated to any actuator which is the case in "thing", such as the switch is controlled when the lamp is commanded to commit a state transition.

- **Observer**. An *observer* observes/monitors the behavior of a physical entity without the possibility to be controlled[8]. To be more general of this capability, it can be used to observe a set of entities or just useful context information, and can be modeled as an FSM, such as the aggregate power monitor on a general electrical counter of the house whose threshold can be offset by external message or event via the grid interface.

Remark 3. The output in the above models (shown in figure 4.2 and 4.3) is denoted by *state* for the graphical clarity reason. It is in fact a set of value assignments $state = \{s_i = v_i\}_{i\in[1...n]}$ where $\{s_i\}_{i\in[1...n]}$ is the set of states of the model and v_i is the Boolean value

[8]In the European project IoT-A (`www.iot-a.eu`), an observer is defined as "anything that has the capability to monitor a Physical Entity, like its state or location".

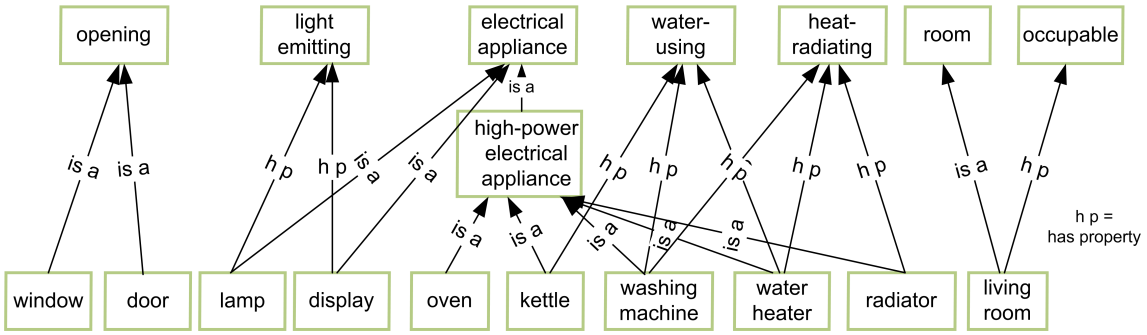


Figure 4.4 – Ontology/taxonomy graph in an home instance example

assigned to each state. In the following of this document, the notation $e.s_i$ represents v_i of s_i at the current state. In the example of the lamp model, the output *state* is $\{off = true, on = false\}$ when it is in *off* state, $lamp.off = true$, $lamp.on = false$. □

These generic models will be used as the *common denominator model* for the leaf categories in the structure described in 4.3.2.

4.3.2 Domain ontology

In the scope of this thesis, an ontology of the concerned domains (home, building, city as application domain of the IoT) aiming to classify target entities is described in the following part for generic control purpose, with the possibility to be enriched by relevant external ontologies, like the ones introduced in 2.4.2.

The primary role of an ontology is to provide categories over entities of interest and their relationships. In our case, the most basic categories of entities are the *well-known* generic categories of appliances or subsets of space: home appliances, rooms, windows, traffic lights, street, etc. Another way to categorize them highly relevant to the purpose of their representation in a generic infrastructure is by properties that may cut cross primary categories. In Figure 4.4, the property *light-emitting* category includes not only the category *lamp* whose primary functionality is to emit light, but also, less obviously, all categories which emit light by side effect of their primary function, like LCD screen. Figure 4.5 shows a part of an example of city ontology, which is also possible to be combined with the home ontology example to form a more comprehensive one, where a property *occupable* category captures the common characteristic of street, lane and residential rooms, and property *light-emitting* for street light, urban signage and lamp, display.

The categorization of entities from such generic features is structured as a DAG which is a representation of the domain specific ontology. This graph makes it possible to follow a path from the most generic ancestor categories (closer to the root nodes) to the most specific descendant ones (closer to the leaf nodes), which establishes a hierarchical relationship. According to the received interpretation of a DAG, descendants d inherits properties from or "traits"/interfaces their ancestors a following an upward path of $d\ Is\_A\ a$ ("is_a" on the edge) or d *has property* a ("h p " on the edge) relationships.

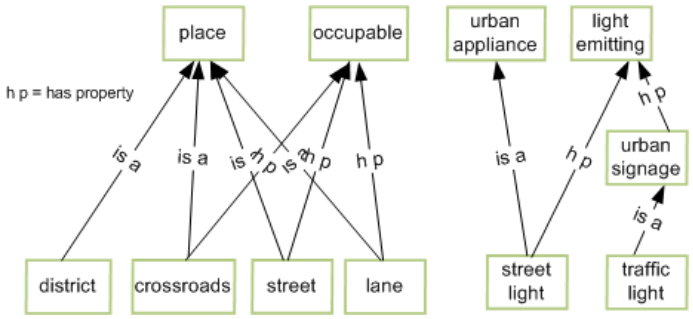


Figure 4.5 – Partial ontology of city

The two underlined relationships are not always absolutely distinguishable, and are in some cases interchangeable. For example, in the current ontology graph, the *electrical appliance* can be considered as a generic type of *display* following the relationship "is_a" (a *display* device is an *electrical appliance*), or as a property of *display* following "h.p." (a *display* device consumes electricity). Both interpretations make sense with nuance in terms of interpretation. A common criterion to distinguish them is that the transitions in a "property" FSM are often abstract and not-directly-controllable, e.g. light, heat, whereas the "classical" taxonomic FSM's transitions react to concrete and controllable event, like electrical switch position change. Therefore, we will consider that one descendant category "has only **one**" "subsumption" ancestor whereas it may "has **multiple** properties". For readability and simplicity of notation, in the following part, we will use the predicate *hasParentStateMachine* to denote both relationships if no distinction is necessary, and note it as $[child : hasParentStateMachine : parent]$.

Some advantages of such way of structuring generic (concrete or abstract) characteristics of the physical world into an ontology are the following:

- This DAG can be easily interpreted as a graph database to get the full benefits of its high accessibility and other supported properties of a normal graph [RWE13].

- It is possible to automatically attribute features to concretely existing object types by traveling the graph, comparing to the more classical and common practiced method which consists of making a explicit declaration for every feature. For example, if we complement the current ontology by associating a sub-DAG to one or several nodes, the entry node(s) will automatically inherit the properties of the ones they attach to, so as their children.

- By contrast to functional groups made manually by declaration in configuration file and static during runtime like in openHAB (cf 2.3.4.2) which is a tedious procedure when the number of entities to be grouped increases, groups can be automatically constructed according to the nodes in the ontology. By doing so, a type of entity with multiple physical aspects may be included in different groups which are managed independently by different services. Dynamic changes of group members during runtime are also possible by the dynamic nature of the group assignment method.

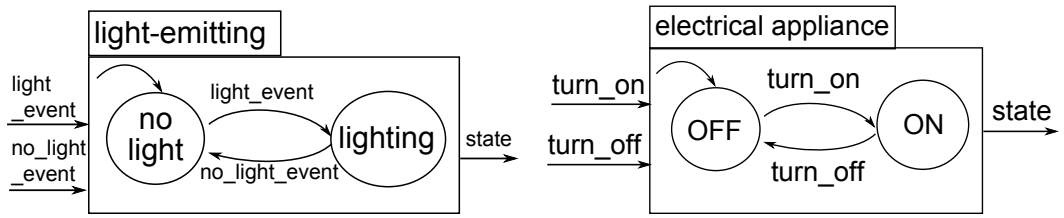


Figure 4.6 – Generic model: (a)light-emitting; (b)electrical appliance

4.3.3 Virtual entity model

Virtual entities (*VE* in the following) refer to the nodes, i.e. the general categories including the leaves, in the ontology DAG. Each VE has a common denominator finite-state model of its own which is meant to be as generic as possible for its features, in order to model all the entities, with more specific features or not, which belong to this category. For example, the general *light-emitting* category has a finite-state model with two states *lighting* and *no-light* (figure 4.6(a)), and *electrical appliance* category has two basic states *off* and *on* (figure 4.6(b)). The leaf nodes in the DAG are considered as the most specific categories in the ontology without any descendant. The generic entity models described in 4.3.1 are maintained by the framework as the common denominator model for the leaf nodes.

The VE model, i.e. the common denominator finite-state model associated to each node in the DAG, has two main uses:

1. **Identification template**. A physical entity in the environment should be firstly identified by the system to be then controlled, which makes the identification an essential step. According to available information in the possession of the system (sensor data, discovered relationship with other entities, id code provided by the entity itself, etc), the identification can be precise or approximate. Some behavior patterns can be extracted from observed data, for example, the entity has 3 distinct functional states and emits heat to be determined as a radiator. The VE model which matches the patterns the most will be taken to model this entity, and the entity is considered as instance of the virtual entity until some more precise identification is made.

2. **Intermediary abstraction**. Though the most exact model of an entity instance depends on the identification process, it can be considered, according to the monitoring/control objective, as an instance of its own model's ancestor without creating a new instance of this model, by omitting some objective-non-relevant properties of the more exact model. This intermediary abstraction helps reduce the system model's complexity in case of no need of precise model.

Remark 4. In case of need of very specific model, the framework does not forbid to use more detailed model which is not provided, but the genericity and the capability of using ancestor models will be lost. □

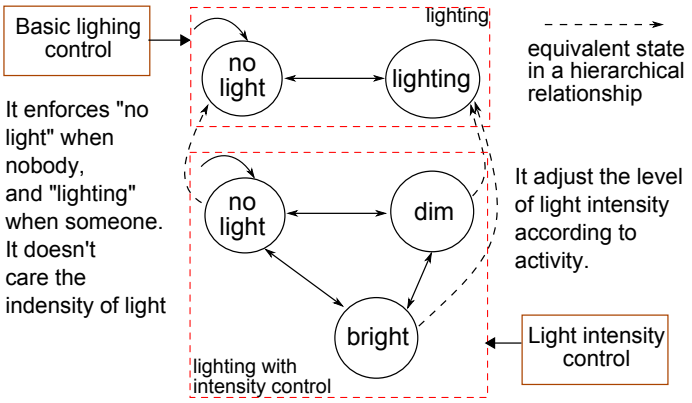


Figure 4.7 – Example of using different abstraction level models

4.3.4 Establishing hierarchical relationship between virtual entity models

In order to practically use a more generic FSM model instead of a more specific one in case of general level monitoring/control on instances' states, the states should be carefully mapped between ancestor and descendant models to keep every operation on one abstraction level valid on the other level as well. Taking an example of the indoor light control consisting all the light-emitting (natural or artificial) instances in the room, including the three-level-controllable bulbs (Philips Hue kit for example) with 2 levels of brightness when they are on. When the control needs to cut all light, all the instances should go to the *no-light* equivalent state, including the three-state bulbs to their *off* state, whereas when the lights are allowed to be on, three-level bulbs can choose to stay either in *dim* or *on* state equivalent to *lighting* state in the general model. Some state changes coming from other sources (user pushed the switch for example) involving transition between equivalent states in the ancestor model will be detected by the control, e.g. from `dim` to `off`, while the others make no difference to it, e.g. from *dim* to *on*. This example is illustrated in figure 4.7.

Which states to be mapped? Hierarchical model pairs following either the basic subtyping relationship "is-a" or the protocol-like relationship "has property" map their states according to the physical meaning presented by the state. For example, by common knowledge we know that a lamp emits light when it consumes electricity, thus a link is established to connect *lighting* in the model of *light-emitting* and *on* in *lamp*. The state *off* and *standby* in *washing machine* both consume no electricity (the very low power consumption in *standby* is ignored in our scope of application), it is obvious to map them with the state *off* in its ancestor *high power electrical appliance*. The mapping is directly declared by the states in the child model, and is considered as one part of the following state machine ontology illustrated in figure 4.8, based on the ontology proposed in [Dol04], with the predicate *hasMappedState* declaring its corresponding state in one of its parents. In the following, we will adopt the notation **[childName].[stateName]** ***hasMappedState*** **[parentName].[stateName]** to denote this hierarchical relationship between two states.

Note that the mapping of states only makes sense when they are respectively in FSMs in a hierarchical relationship described by the current ontology. Thus, within the present

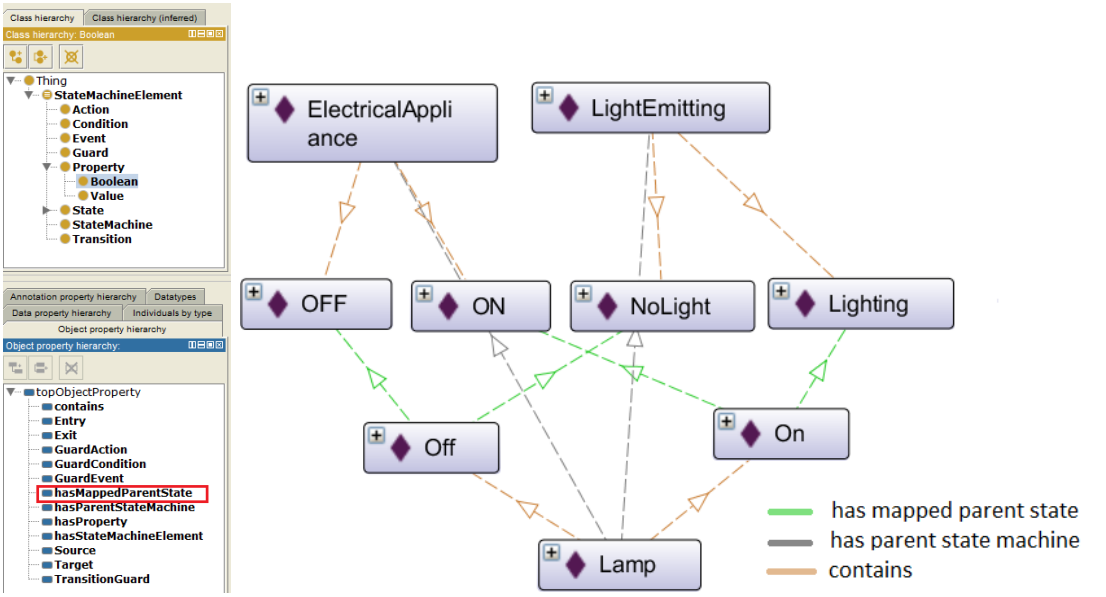


Figure 4.8 – (a)State machine ontology; (b)Example of state mapping of *light-emitting* and *lamp*

thesis, when we talk about sub-/super-state, we mean implicitly that they are states of a pair of descendant/ancestor FSMs.

The states between the already known ancestor/descendant models should to be mapped in the way that state transition in the descendant model would never go out of the state space of the ancestor, and every state of the ancestor model can always find its corresponding state in any of its descendants. The choice of such assumption may be arbitrary. Nevertheless, there is one important reason to make us make this choice to keep our modeling framework general and consistent instantaneously. A specific model at the bottom or the lower part of the DAG should be at every moment able to be considered as its ancestor which is more generic, for generic monitoring or control purpose. While it works as a specific modeled entity going to a state having no corresponding state in its ancestor model, and this ancestor is one of the subjects of a general control objective, the underlined control has no possibility to accomplish the objective as this entity is in an unknown state to it. To avoid such control failure due to inconsistency in state mapping, a strong hypothesis of descendant staying in ancestor's state space is necessary.

This mapping procedure (declaration of mapped states in parent models by each state of the child) is done manually for every parent/child pair and is just one method among the others which are valid as long as they satisfy the above assumption, without the need of knowledge of the method's details.

How to map two states? The principle of the proposed method is to consider ancestor/descendant models as hierarchical state machines where every state in a descendant model is a substate or nested state of a state in its ancestor. The substate inherits the entire behavior (actions and property values to take) of its superstate like in the semantic of Heptagone, if a state S_n of a sub-automaton nested in the superstate S_s is active, both

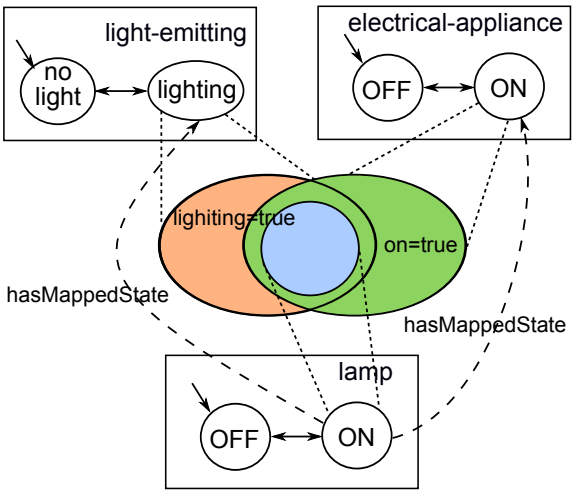


Figure 4.9 – *Multiple inheritance*: lamp.on inherits values from parents

S_n and S_s are considered active, thus the result behavior of S_n to an outside observer is as if S_n has it own specialized behavior plus all the behavior of its superstate S_s. This *behavioral inheritance* [SM00] in hierarchical state machines is very similar to the concept of *inheritance* in Object-Oriented programming (OOP). Substate is analogue to subclass in OOP which only needs to implement the differences compared with its superclass whose entire properties are inherited by the subclass. If there is no proper behavior to the state of the descendant model, the state's behavior is processed and obtained automatically: it inherits the behavior of all its parents, similar to the concept of *multiple inheritance* in OOP. It should be marked here that behavioral inheritance describes the hierarchical relationship between sub-/super-states and should not be confused with state machine inheritance.

Taking the example of *lamp* which is child of *lighting* and *electrical appliance* at a time. As declared explicitly in the model of *lamp*, lamp.on *hasMappedState* {light-emitting.lighting, electrical-appliance.on} and lamp.off *hasMappedState* {light-emitting.no-light, electrical-appliance.off}, lamp.on inherits the value "lighing=true" from light-emitting.lighting and "on=true" from electrical-appliance.on, and lamp.off inherits the value "no-light=true" from light-emitting.no-light and "off=true" from electrical-appliance.off. As result, lamp.on has 2 inherited values from parent states as illustrated in figure 4.9.

Following this method of state mapping, we notice that from the top-down point of view, a descendant is a *refinement* of its ancestor, and from the bottom-up point of view, an ancestor is an *abstraction* of its descendant in the level of state [Har87]. Every state of a topmost model in the ontology DAG consists of one or several descendant states. Every state of a bottommost model is inside of one state of an ancestor. Every state of a mid-level model is at a time inside of one ancestor state and consists one or several descendant states. This is illustrated in Figure 4.10 showing a branch in the ontology DAG consisting *electrical appliance*, *high power electrical appliance* and *washing machine*. The states *washing*, *rinsing*, *spinning* all inherit the behavior "keep electricity consumed" of state *on* in *high power electrical appliance* which is identical to *on* in *electrical appliance*. This also ensures the transitivity of behavior inheritance from superstate which is many levels away.

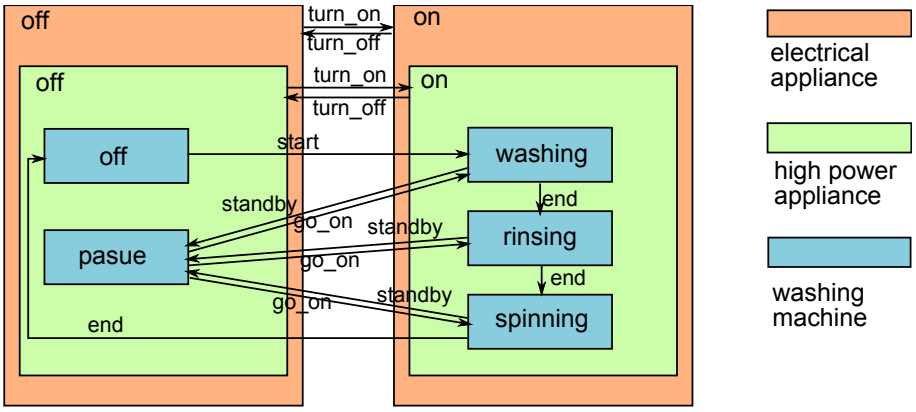


Figure 4.10 – Ancestor/descendant state mapping in "is a" relationship

Multiple inheritance issues. If two parents both have one property but with different value, ambiguity or value conflict on this property occurs when a child inherits both of them[9]. Some constraints should be added to the state mapping which will be checked for the validity before use. Consider that B and C are children of A, and D is child of both B and C. Assume that in our scope of application, inherited properties are not modified along its life because the physical meaning behind stays the same. It should be guaranteed that if a property exists in D, it cannot have two different values inherited respectively from B and C. The property in question can be expressed as following:

$$\{p \in (propsB.key \cap propsC.key) \mid (\text{"}p = x\text{"} \in propsB) \wedge (\text{"}p = y\text{"} \in propsC)\}$$

where:

- p is the name of the property[10];
- $propsB$ and $propsC$ denote respectively the set of properties B and C;
- them followed by $.key$ denote the set of names of their properties;
- x and y are the values of the property

It should be made sure that $x = y$ in any situation. As:

$$\begin{aligned} &propsB.key \cap propsC.key \\ &\quad = (propsBown.key \cup propsA.key) \cap (propsCown.key \cup propsA.key) \\ &\quad = (propsBown.key \cap (propsCown.key \cup propsA.key)) \cup (propsA.key \cap \\ &\qquad (propsCown.key \cup propsA.key)) \\ &\quad = ((propsBown.key \cap propsCown.key) \cup \underbrace{(propsBown.key \cap propsA.key)}_{\emptyset}) \cup \end{aligned}$$

[9] It is referred as the "diamond problem"

[10] A *property* is a pair in the form "$propertyName = propertyValue$"

$$\begin{aligned}&(\underbrace{(propsA.key \cap propsCown.key)}_{\emptyset} \cup \underbrace{(propsA.key \cap propsA.key)}_{propsA.key})\\ =&(propsBown.key \cap propsCown.key) \cup propsA.key\end{aligned}$$

where:

$propsA$ is the property set of A; $propsXown = propsX - propsA$ and by assumption that a child does not override a parent's property, $propsXown \cap propsA = \emptyset$.

In order to guarantee $x = y$, we should only to make $propsBown.key \cap propsCown.key = \emptyset$ because by definition, if $p \in propsA.key$, it takes a unique value defined in A, which implies that B **and** C **should not have a same property other than the ones in their common parent**.

4.4 Proxy instance plane

The word *proxy* is used as its literal meaning in the context of the thesis: a representative for someone/something else. Located in the middle of the physical world and the abstract world on paper, this plan aims to "represent" the physical world by creating individual "representatives" based on "template" noted on the paper.

This plane captures and maintains direct, instantiated informational representations of entities from the physical plane, drawing upon generic models from the model plane to provide a shared interface of the physical world to applications. The system architecture has five layers depending on different abstraction level and function, from the device proxy layer to the application layer. It takes the ROA approach where each proxy and their current state (of device or of entity) is considered as a resource which provides the information in accordance with the current internal semantics via a uniform interface like a URI.

4.4.1 Device abstraction layer

Devices are taken in the same sense as in 4.2.3 with focus on their upper level interface independent of the protocols used to connect them and of the technology details of their actual implementation, as proposed by various standardization bodies. They are virtualized and used through their proxy which is an "object" regrouping accessible software interface variables of the device. For example, the proxy of a smartPlug is a set of upward data interface of instantaneous power reading and downward interface of "turn on/off" action sending. The mapping between devices and their proxies is one to one, unless these devices are never used separately (e.g. an array of identical sensors used for array processing).

4.4.2 Entity instance proxy layer

This layer provides a one to one mapping between physical entities and a set of proxies that maintain their states by instantiating corresponding models from the model plane.

An entity proxy maps directly to a corresponding node in the model plane that provides a template, as an object maps to a class in OOM. It provides an interface both to relevant devices to get actual sensor readings or forward control orders to actuators, and to higher layers of the infrastructures to provide information about current state and relevant variable values. Several instances of the same automaton model can co-exist.

In this layer, entity instance proxies may have relationships other than the hierarchical ones captured from the model plan, e.g. the relationship *located in* links a lamp proxy and a room proxy if the lamp is in the room, the relationship *owned by* links several object proxies and a person if the objects belong to the person. These relationships are purely dedicated to the current environment that we call them *extrinsic relationship* against *intrinsic relationship* referring to environment-independent ones. Entity instance proxies together with the *extrinsic relationships* connecting them make a *graph of instances* that is maintained in the infrastructure and ready to be explored.

4.4.3 Entity group layer

While virtual entities (VEs) are a categorization of identified physical entities by a specific type or by an intrinsic property which have a common denominator behavior model, entity group models associated to a VE represent the collective behavior of all the entities modeled by the underlined VE model since one of the original purposes of the classification is to manage a set of entities together instead of each one individually. Thus they involve the composition of several identical individual VE models. More details of the models for entity group made by entities having identical VE model will be given in 5.1.

Also, an entity group can present a group constituted following an *extrinsic relationship* recognized and maintained in the entity instance proxy layer. However, il would be difficult to find a common denominator model for this kind of group because they do not necessarily share any common behavior.

Entity group models can be dedicated to one purpose of use, and one VE could have more than one instantiated entity groups based on different models. For example, both on a group of radiators, an instance of entity group model containing a state *some radiator in frost protection mode* would be interesting for a safety control scenario (preventing frost danger in pipes), but totally insignificant for a comfort control scenario whose goal is to maintain the temperature at 24 degree. If in an environment instance, the above safety and comfort objectives coexist, each of them uses the entity group instance of its corresponding model, thus 2 different entity group instances exist which are associated to the same VE.

If we consider the entity instance proxies which have a 1-to-1 mapping to physical entity as something "real", an entity group can be considered as a *view* of a set of entity instance proxies which is something "abstract". Thus different *views* are as direct consequence as they are taken from different *points of view*, e.g. different objectives.

Connect entity groups and entity instance proxies. Associations between entity groups and entity instance proxies depend on both intrinsic and extrinsic relationships,

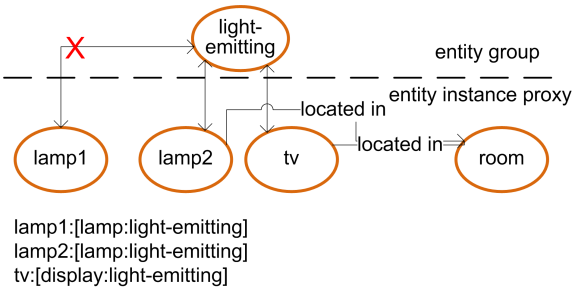


Figure 4.11 – Association is made on both intrinsic hierarchical relationship and extrinsic environment-specific relationship

i.e. the hierarchical DAG in the model plan and the graph of instances maintained in the runtime environment which respectively does a filtration of existing entity instance proxies for a given entity group according to entity type and perimeter/scope of application. Both relationships are indispensable: only the category that the entity group is associated to or its sub-categories may provide compatible inputs for entity group and react correctly to the outputs; only one specific part of all the entity instance proxies of the category is interesting for one given application. Unlike the category relationship which is directly reflecting the model plan, the necessary extrinsic relationships are specified in a configuration file of the given environment instance. In the example of indoor light control in 4.3.4 addressing to *light-emitting group* (in figure 4.11), the mapping between the group and entity instance proxies is done by taking the intersection of all the proxies whose model is descendant of or is *light-emitting* itself and the ones which is connected to the target *room* by a *located in* relation in the graph of instances.

Observer as a group. In 4.3.1, an observer is a monitor on a physical entity or a piece of context information which is often used to determine if a condition is satisfied to trigger an action. It is more logic to consider them instantiated in the entity group layer as: first, in the current infrastructure, physical entities are already monitored by their proxies in the entity instance proxy layer, which implies that there is no need to have a second observer for each of them individually and if there are observers, they are about information directly from sensors without physical entity support, like the temperature; second, a context property is most time contributed by more than one physical entity, which can be considered as a "collective behavior" of the contributor-entities; third, their result of observation is often used directly by services which try to interface only with entity group layer.

4.4.4 Service and local application layer

This layer hosts services and other applications that are tightly coupled to the infrastructure. They need more knowledge of the semantics of entity models and may also mandate bounded latency constraints between system responses and actions. They can read all information from output of state model instances and perform actions directly on the available inputs of the state models. Examples of services and applications in this layer are local light control (quick reaction to turn on light when presence is detected arrives),

real-time electrical load shedding control from detection of overload.

This layer does also provide north bound interface to remote loosely-coupled applications which have less strict temporal constraints and may be content with a "best-effort" REST interface through a wide-area network.

4.4.5 Remote applications layer

Most monitoring and control applications that are loosely coupled with the physical plane are not supposed to be part of the proposed infrastructure, or to be co-hosted with it. The infrastructure is used to separate them from the particulars of sensors, actuators, devices and other pieces of equipment that they operate on. As represented in Figure 4.1, such applications operate on top of the instance plane, which provides them the required interface to observe and control the states of individual entities through their proxies, or the entity groups they belong to.

Chapter 5

Generic models for discrete control based on the shared infrastructure

Contents

5.1 Control-oriented models

We have seen generic FSM models for physical entities and even more abstract models for intermediate categories in the domain ontology presented in 4.3.1. These models capture the essential states and possible actions of the modeling target entities according to their nature. However, if we want to use DCS technique directly on these models, controllable variables which represent the control decision from the discrete controller should be identified on these models, so that some transitions between states can be prohibited or enforced to make the entire system stay in a legal state. Also, as we introduced in 4.4.3, entity groups are made to make it possible to manage a set of physical entities sharing the same property together instead of managing them individually in order to have a higher level of abstraction and less complexity, especially to avoid the combinatorial explosion of state space due to the nature of DCS method, DCS should also apply on the models of entity group, which implies that these models should be FSM with controllable variables on their transitions[1]. We suppose that the actions triggered by each transition or in each state are effective that no asynchronous verification for the effectiveness is needed.

[1]We do not deal with the case where group is constituted by extrinsic relationship because there is no common denominator model as explained in 4.3.3.

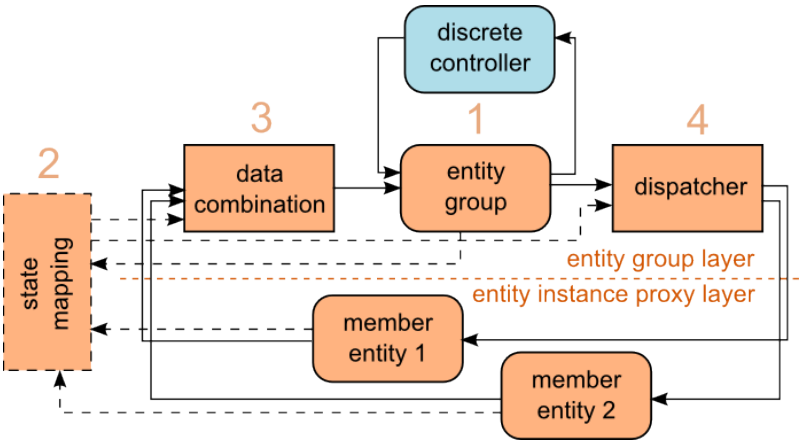


Figure 5.1 – High level overview of entity groups and entities with functional supportive blocks

This section will be organized following the number given in figure 5.1: 5.1.1 will describe the template for forming an entity group model; 5.1.2 will explain how to establish the state mapping between the entity group and the entities belonging to it so that in 5.1.3 the input of entity group in terms of logical combination of states of individual entities can be calculated; 5.1.4 will give more details about how a control order from the supervisory controller on the entity group will arrive at individual entities which need to be actuated.

Vocabulary definition . Before going into details of construction of entity group models and working mode of entity group instances, we introduce some vocabularies that are applied during the entire explanation, with the help of the a "toy" example illustrated in figure 5.2.

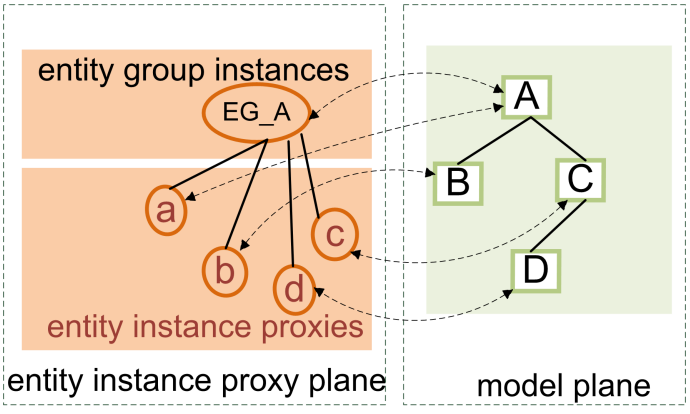
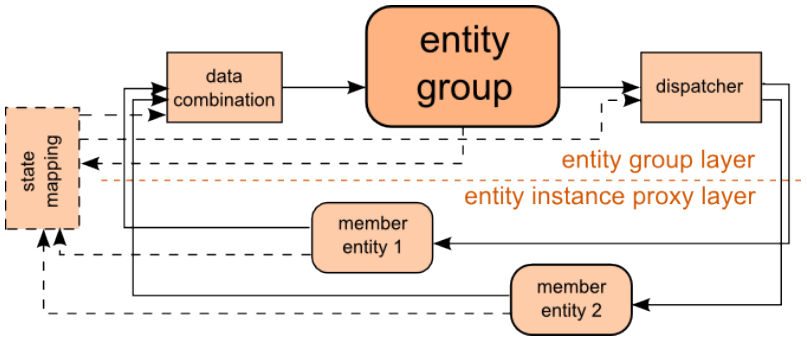


Figure 5.2 – Overview of the relationship between entity group and individual entities via the model plane

- **Entity group**. It is located in the entity instance proxy plane, and is an instance of one **entity group model** associated to the **group category**. In figure 5.2, `EG_A` is the entity group associated to the **group category** `A`.

- **Group category**. It is short for "the category in the ontology DAG in the model plane used to defined an entity group in which all individual entities have the shared property of the category". In figure 5.2, `A` is the **group category**.
- **Group category VE**. It refers to the VE model (see 4.3.3) of the **group category**. In figure 5.2, it means the VE model of `A` which is not shown.
- **Member entity**. A member entity of an entity group is an individual entity, known as an entity proxy, which is an instance of the VE model of the **group category** itself or one of its descendant categories. In figure 5.2, group category `A` has `B`, `C` and `D` as descendant categories. Thus, entity instances `a`, `b`, `c`, `d` which are respectively instances of VE model of `A`, `B`, `C`, `D` are member entities of entity group `EG_A`.

5.1.1 Entity group models



One entity group model is one perspective of seeing how several individual entity instances behave together. According to different objective (monitoring or control), the manner of abstracting a complex collective behavior may differ a lot from each other to have just the necessary information to keep the system as simple as possible. An example is given in 4.4.3 about different models of a group of radiators for different goals. Thus, no entity group model is directly associated to a node in the ontology DAG in the model plane as it varies to meet the correct level of abstraction according to objectives. However, absent of any more specific model, a very generic and basic template provided by the framework may be used as a modeling tool for entity group model of a general category representing multiple entity instances which could be modeled by the VE model of this category.

Remark 5. This generic template given and described in the following is just **one** choice among many others, which has already been mentioned in 4.4.3. For example, it is useful to consider a state of "one street lamp is on every two lamps" for energy efficiency management, which does not exist in the template we present. Nevertheless, we think our template is useful in a lot of cases related to the current application scope, Home, and it presents a correct trade-off between reduction of complexity and reservation of useful information related to the type of control we intend to do, basic and supportive discrete control which stay valid in most of the environment instances in the domain. □

The generic template is designed to reflect all the possible combinations of the group category VE's states in which member entities are in. A state of the template is distinguishable from others when the set of active states of the entity group VE is different. It

takes as input the logical conjunction (**and**, noted by $\bigwedge$) and disjunction (**or**, noted by $\bigvee$) of the Boolean variables indicating the member entities' current states to determine the current group state which is, as an output, interpreted as a logical combination of member entities' states in which they are or should be in. The result of logical combination of Boolean values does not depend on the number of passed variables, but on the truth value of each variable and the applied operation.

Property 1. *The number of states in the entity group model using the above template is* $N = \sum_{k=1}^{n} C_n^k = 2^n - 1$[2]*, n is the number of states of the group category VE.*

Explanation. Let $S = \{s_i\}_{i\in[1...N]}$ be the set of states of the entity group, $S_{VE} = \{s_{VEi}\}_{i\in[1...n]}$ the set of states of the group category VE model. By construction, s_i is defined as a subset of S_{VE} representing the distinguishable states of the group category VE present in all its member entities. The problem of finding S is then reduced to finding all the subset of S_{VE} ($\emptyset$ excluded). The number of combination of k distinct elements chosen from a set of n elements is C_n^k, where $k \in [1 \dots n]$. So for all the possible values of k, the total number of subsets of S_{VE} is $\sum_{k=1}^{n} C_n^k$. □

The number of input variables of the entity group model is $2n + 2$, where $2n$ variables are $\bigvee e.s_{VEi}$ and $\bigwedge e.s_{VEi}$ (the definition of $e.s_i$ is given in 3) and 2 controllable variables c_1 and c_2, one for inhibition and the other for enforcing. We will see further in the chapter that the number of input variables can be reduced if the relationship among elements of S_{VE} is known.

Remark 6. $\bigvee e.s$ is a simplified notation for $\bigvee_{i\in[1...n]} e_i.s$ which denotes the **OR** operation, and the $\bigwedge e.s$ denotes the **AND** operation. The initial state of the entity group model based on this template is the one where all the member entities are in their initial state. □

The significance and the entering condition (combined with controllable variables to constitute the complete guard of the transition towards the state) for each state s_i of the entity group model should be expressed in terms of the logical combination of the states of all the member entities. For doing so, we distinguish 3 cases:

1. s_i represents that all the member entities in the same state s_{VEj}, which can be interpreted easily as
$$E1 = \bigwedge e.s_{VEj}$$
There are in total n (C_n^1) states of the entity group model in this case.

2. All the states in the entity group VE model are present in s_i, which can be interpreted as at least one member entity in every s_{VEj}. In Boolean expression,
$$E2 = \bigwedge_{j\in[1...n]} (\bigvee e.s_{VEj})$$
There are in total 1 (C_n^n) state of the entity group model in this case.

[2] C_n^k is also equivalent to the binomial coefficient denoted by $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$

3. s_i represents that some s_{VEj} are present and some are absent. If we define the present states are $S' = \{s_{VE\_present\_k}\}_{k\in[1,m]}$ and the absent states are $S \setminus S' = \{s_{VE\_absent\_l}\}_{l\in[1,n-m]}$ where $1 < m < n$. For those states which are present, there is at least one member entity in this state, expressed as $(\bigwedge_{k\in[1...m]}(\bigvee e.s_{VE\_present\_k}) = true)$, and for those states which are absent, there should be no member entity in this state, expressed as $\neg(\bigvee_{l\in[1...n-m]}(\bigvee e.s_{VE\_absent\_l}))$. The complete

$$E3 = \bigwedge_{k\in[1...m]}(\bigvee e.s_{VE\_present\_k}) \bigwedge \neg \bigvee_{l\in[1...n-m]}(\bigvee e.s_{VE\_absent\_l})$$

The following algorithm 1 summarizes the above procedure:

Algorithm 1 Significance and entering condition of entity group model state in terms of group category VE state combination

1: let $S_{VE} = \{s_{VEi}\}_{i\in[1...n]}$ be the set of group category VE model states
2: **for** every entity group model state s **do** ▷ N states in total
3: let $S'_{VE} = \{s_{VE\_present\_k}\}_{k\in[1,m]} \subseteq S_{VE}$ be the set of group category VE model states present in s, $1 \leq m \leq n$
4: **if** $m = 1$ **then** ▷ all the member entities in the same state
5: $s \leftarrow E1$
6: **else if** $m = n$ **then** ▷ all group category VE states are present
7: $s \leftarrow E2$
8: **else if** $1 < m < n$ **then** ▷ some group category VE states are present, some are not
9: $s \leftarrow E3$
10: **end if**
11: **end for**

Remark 7. The procedure automated by the algorithm is about generating a model which could be maintained in the same format as other VE models in the DAG. The entity group used in the proxy plane is the instance instantiated from the generated model just as the entity proxies instantiated from their corresponding VE models. □

The simplest case is the entity group template for a set of instances of a 2-state FSM model with states `X` and `Y` as illustrated in fig 5.3(a). This model takes `a` and `b` as input to enable a transition, and outputs the value of Boolean variables `x` and `y`, where $\{x = true, y = false\}$ for state `X` and $\{x = false, y = true\}$ for state `Y`. The resulted group model has thus 3 states: `All X`, `X or Y` and `All Y` as all the possible combinations of `X` and `Y`. The useful inputs enabling the transitions between such states are as shown in fig 5.3(b). The last two input variables `c` and `c'` are controllable whose value is given by the controller in order to enforce or inhibit the transition. The output variable value indicating the current state of the group is interpreted as the result of the logical combination of member entities' state value that they should make. In this case, $All\,X = \bigwedge e.x$, $All\,Y = \bigwedge e.y$ and $X\,or\,Y = \bigvee e.x \wedge \bigvee e.y$.

Property 2. *With a 2-state FSM where the true value of two Boolean variables* `x` *and* `y` *indicates respectively the active state* `X` *and* `Y`*, the expressions on e.x (or e.y) can be replaced by e.y (or e.x):* $\bigvee e.x = \neg(\bigwedge e.y)$ *and* $\bigwedge e.x = \neg(\bigvee e.y)$

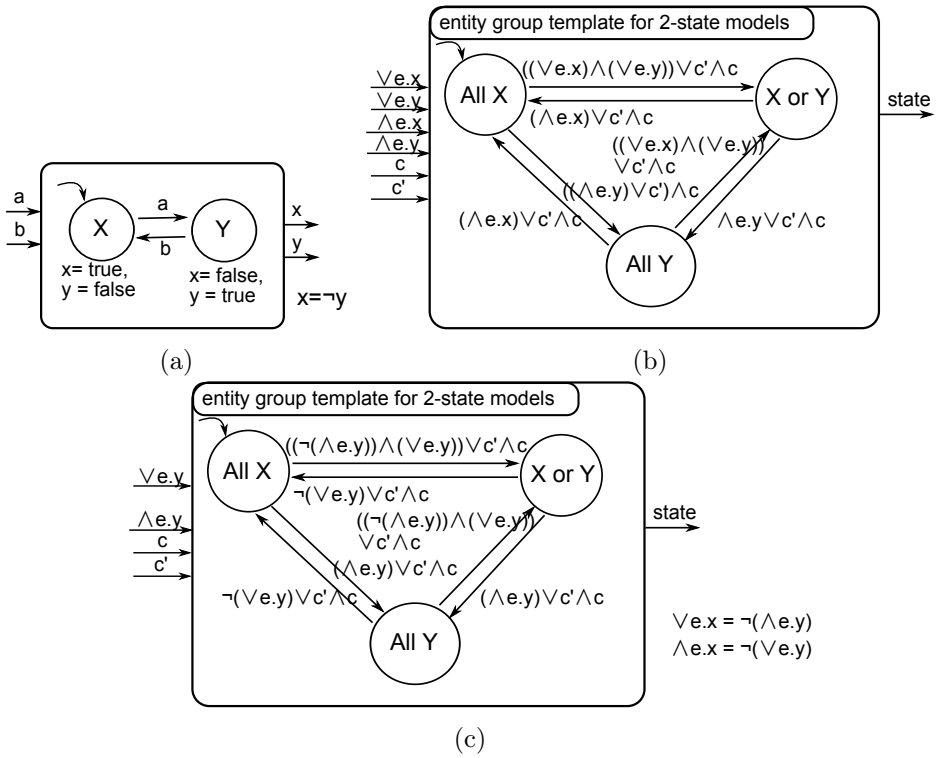


Figure 5.3 – (a) 2-state VE model; (b) general template for group model of 2-state models and its simplified version (c)

Proof. We know that $X = \{x = true, y = false\}$ and $Y = \{x = false, y = true\}$, thus $e.x = \neg e.y$. So we have:

$$\bigvee e.x = \bigvee(\neg e.y) = \neg(\bigwedge e.y)$$

$$\bigwedge e.x = \bigwedge(\neg e.y) = \neg(\bigvee e.y)$$

□

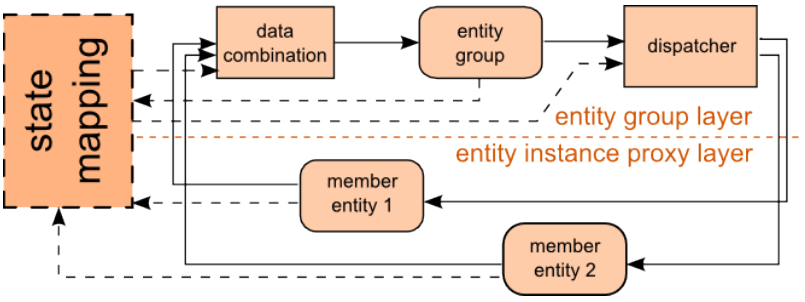
The above property 2 makes the simplified entity group template for 2-state VE models as illustrated in 5.3(c).

Advantage of control on group models Entity group models are the ones provided to the upper level services, especially the synthesized controller service, as the system's behavior model. The advantages of managing entities by group are especially:

- Flexibility in configuration change. The entity group hides completely the individual entities from upper level services or applications that any change in entity instance proxy level, e.g. a new extra radiator in the room during winter, is transparent for them. Only an update of mapping between present entity instance proxies and their group is solicited to get the correct input value for the entity group and to transmit the output value to a valid entity instance proxy, instead of a re-design or re-compilation of services.

- Reduction of state space for the procedure of synthesis of controller on top of these system behavior models. The state space increases exponentially to the number of models. Even the parallel composition of the most simple 2-state models would lead to a state explosion when the number of automata grows. A model with states representing the significant collective behavior after the parallel composition is one solution to reduce the state space that the DCS should deal with. A every simple example can make it very obvious. Taking the first control objective in the current example scenario with n 2-state *light-emitting* instance proxies under control. The DCS performed directly on the individual instances gives a space of $\mathbf{2}^{\boldsymbol{n}}$ states, while the one on the entity group model gives **3** states with assumption that the group model adopts the basic 3-state template for composition of 2-states state machines. When $\boldsymbol{n}$ goes up, this difference becomes significant. A simple comparison will be given in 7.2.4 to illustrate this advantage.

5.1.2 State Mapping of group category VE and member entities



An entity group model, no matter being specific to a given objective or using the generic template, is always expressed in terms of the group category VE model and its states, while its member entities are actually instances of more specific descendant categories. From the point of view of the entity group instance, all the member entities are as if they were instances of the group category VE. This **over**-abstraction is enabled thanks to the state mapping explained in 4.3.4 guaranteeing the compatibility of using an ancestor model in lieu of a descendant model which has actually instantiated the recognized entity. The parent(ancestor)/child(descendant) relationship between categories in the ontology is used to determine which individual entity instances are to be included by the entity group, as already explained in 4.4.3. And the state mapping between the group category VE and the member entities is used to determine the right input value of the group and the correct output value interpretation in terms of child model's state if necessary.

Remark 8. In the above illustration we notice that the "state mapping" module is delimited by dotted line, different from the others. This is a "semi"-static module in the runtime: the information held does not change as long as there is no configuration change such as new added member entity, more detailed and specific entity type identification. This module does not directly participate the dataflow within the closed loop made by other modules, however, it verifies the configuration updates and is consulted by "data combination" and "dispatcher" blocks to ensure the right interpretation in both upwards and downwards directions. □

To get the state mapping correctly and efficiently, two steps are needed: (1) map the

states of the group category VE and the member entity's VE; (2) map the states of group category VE instance (what a member entity is seen by the entity group) and the member entity using the result of the first step. We will explain the 2 steps in the following with the help of an illustration in figure 5.4 which is a detailed version of figure 5.2, to finally get an integrated algorithm which does the procedure for a given entity group and its member entities in the system runtime.

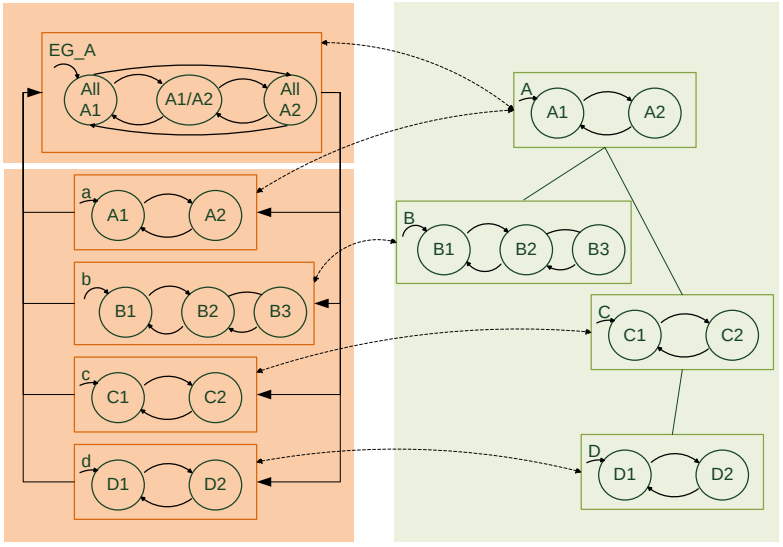


Figure 5.4 – Illustration for state mapping procedure: developed from figure 5.2

Step 1: state mapping between the group category VE and member entity VEs. Though all the information about state mapping in a pair of parent/child categories is available in the DAG, it is still indispensable of doing this step in order to make the following instance states mapping procedure more efficient for 2 reasons:

- The ontology DAG is an external data structure (database or RDF file) to be accessed with necessary tool. It could be very inefficient if the access frequency is high.
- The state mapping information in the ontology DAG is only expressed explicitly between parent and child. If two categories have intermediary levels between them, the state mapping should be deducted by traveling the path connecting them. It is obviously more quick to store in the runtime memory the state mapping information of ancestor/descendant pairs to avoid retrieving it by traveling the DAG each time required.

The data structure for storing the state mapping information can be a table which is easy to be implemented and used in runtime. The objective of step 1 is to obtain a table as in table 5.1 for the very generate example in figure 5.4, knowing already from the ontology DAG the state mapping of A/B, A/C and C/D. Given a child VE (1st column) and a parent state (2nd column), the mapped child(ren) state(s) can be accessed in the 3rd column.

Parent VE A		
Child VE	Parent State	Child(ren) State(s)
B	$A.A_1$	$B.B_1$
	$A.A_2$	$B.B_2$, $B.B_3$
C	$A.A_1$	$C.C_1$
	$A.A_2$	$C.C_2$
D	$A.A_1$	$D.D_1$
	$A.A_2$	$D.D_2$

Table 5.1 – Table maintaining state mapping of parent/children VE models for a parent.

There are 2 cases to fulfill the table which is initially with only one entry: mapping of the parent VE itself. They are distinguishable by the existence of intermediary categories between the parent VE and the VE to be mapped.

Case 1: direct parent/child. In this situation, it is enough to refer to the information registered in the DAG by invoking the predicate *hasMappedState* of each state in the child VE which gives a list of mapped states in all its parents (cf. 4.3.4). To get the mapped state to the current given parent VE, a filter on the obtained list by the given parent VE is applied (remind that the returned mapped states are in form of *[ancestorName].[stateName]* that the information of the model containing the state is also available). This procedure is detailed in algorithm 2 line 1 as procedure **ParentChildStateMapping**.

Case 2: ancestor/descendant with intermediate categories. In this situation, a path should be determined first from the descendant VE to the ancestor ($D \rightarrow A_1 \rightarrow \cdots \rightarrow A_n \rightarrow A$) before applying several iterations of direct parent/child state mapping process to get the final required state mapping. This path can be found by using basic operation on a generate directed graph that we suppose well known. It is described in the procedure AncestorDescendantStateMapping in algorithm 2 line 12.

Step 2: state mapping between the instance of group category VE and member entities. After the step 1, the table 5.1 is fulfilled for all the present categories in the system runtime, which we can use to find the state mapping between instances following the algorithm 3. In the example in figure 5.4, the result state mapping of member entities is:

$$A.A_1 = \{a.A_1, b.B_1, c.C_1, d.D_1\}$$

$$A.A_2 = \{a.A_2, b.B_2, b.B_3, c.C_2, d.D_2\}$$

The integrated algorithm 4 is composed of the 2 previous steps to finally find the set of entity member states corresponding to each state of group category VE.

Algorithm 2 VE state mapping

1: **procedure** ParentChildStateMapping(P,C) ▷ P is the parent VE, C is the child VE
2: *temp* ▷ a two column table for *parentState*/*childState* pairs
3: $S_P \leftarrow P.getStates()$ ▷ S_P is the set of states of the parent
4: $S_C \leftarrow C.getStates()$ ▷ S_C is the set of states of the child
5: **for all** $s_{Ci} \in S_C$ **do**
6: $listOfMappedStates \leftarrow s_{Ci}.hasMappedState$
7: $mappedState \leftarrow listOfMappedStates.filter(P)$ ▷ get ont the mapped state of P among multiple parents
8: $temp.addEntry(mappedState, s_{Ci})$
9: **end for**
10: **return** *temp*
11: **end procedure**

12: **procedure** AncestorDescendantStateMapping(A,D) ▷ A is the ancestor VE, D is the descendant VE
13: *temp* ▷ a two column table for *ancestorState*/*descendantState* pairs
14: $S_A \leftarrow A.getStates()$ ▷ S_A is the set of states of the ancestor
15: $S_D \leftarrow D.getStates()$ ▷ S_D is the set of states of the descendant
16: $path \leftarrow$findDirectedGraphPath(A,D,graph) ▷ $path$ is an ordered list of intermediate categories from D to A
17: **for all** $s_{Di} \in S_D$ **do**
18: $childState = s_{Di}$
19: **for** $j \leftarrow 1, |path|$ **do** ▷ $|path|$ is the number of elements in the list
20: $parent \leftarrow P_j$ ▷ P_j is the jth element of $path$
21: $childState \leftarrow childState.hasMappedState.filter(P_j)$
22: **end for**
23: $mappedState \leftarrow childState.hasMappedState.filter(A)$
24: $temp.addEntry(mappedState, s_{Di})$
25: **end for**
26: **return** *temp*
27: **end procedure**

Algorithm 3 State mapping of group category VE instance and member entities

procedure InstanceStateMapping(G,e,*table*,*mapping*) ▷ G is the group category VE, e is the member entity, *table* is the complete state mapping table like table 5.1
  $StatesG \leftarrow G.getStates()$ ▷ $StateG$ is the list of states of G
  for $i \leftarrow 1, |StatesG|$ **do**
    $correspondingStates \leftarrow$ getCorrespondingStates($table, e, statesG_i$)
    $mapping.get(statesG_i).add(correspondingStates)$ ▷ added to the set of all corresponding state instances
  end for
  return *mapping*
end procedure

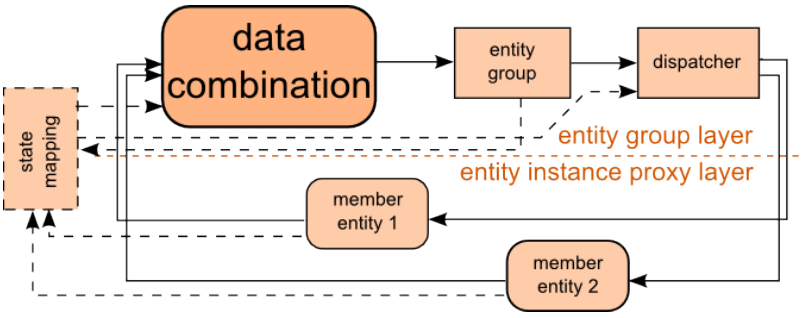
procedure getCorrespondingStates($table, e, state$)
  $E \leftarrow e.getVE()$ ▷ VE model of member entity
  $mappedStates \leftarrow table.get(E).get(state)$ ▷ get the 3rd column of the table for parent state
  $correspondingStates \leftarrow e.getStateInstances(mappedStates)$ ▷ get the state instances of e
  return *correspondingStates*
end procedure

Algorithm 4 State mapping of entity group level VE model and their member entity VE models and instances

let G be the group category VE, $\{e_i\}$ the set of member entities of the entity group attached to G
let *resultMapping* be a 2-column table where the 1st one represents the group category VE state and the 2nd one represents its corresponding member entities' states
table ←CreateTable(G) ▷ creates and initializes a table for this group category if not exist
for $i \leftarrow 1, |\{e_i\}|$ **do**
  $E \leftarrow e_i.getVE()$
  if Entry E does not exists in *table* **then**
    AddEntry(E,G,*table*)
  end if
  InstanceStateMapping(G,e_i,*table*,*resultMapping*)
end for
return *resultMapping*

procedure AddEntry(E,G,*table*)
  path ←findDirectedGraphPath(G,E,graph)
  if $|path| = 0$ **then** ▷ there is no intermediate category
    ParentChildStateMapping(G,E)
  else
    AncestorDescendantStateMapping(G,E)
  end if
end procedure

5.1.3 Data combination

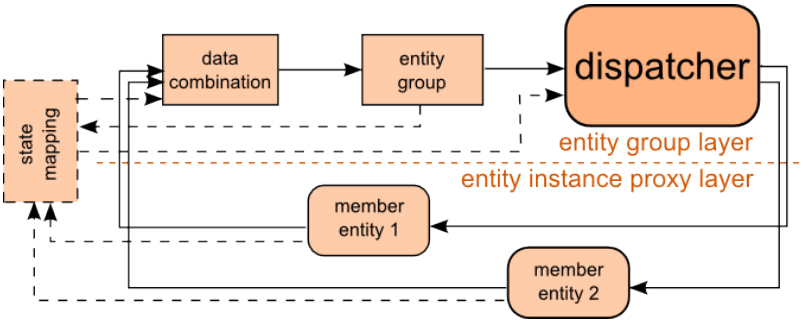


Having the result from previous subsection, the input of the entity group can be expressed in terms of member entities' state by replacing the group category VE state by its corresponding state in the member entity. In the example in figure 5.4 where we consider the entity group model has been built from the generic template we introduced in 5.1.1, its input, initially expressed all in states of A, can be expressed in member entities' (a, b, c, d) states:

$$\bigvee e.A_2 = a.A_2 \vee b.B_2 \vee b.B_3 \vee c.C_2 \vee d.D_2 = \neg(\bigwedge e.A_1)$$

$$\bigwedge e.A_2 = a.A_2 \wedge b.B_2 \wedge b.B_3 \wedge c.C_2 \wedge d.D_2 = \neg(\bigvee e.A_1)$$

5.1.4 Control order dispatching towards individual entity instances



So far, we have seen how to make an entity group model and how to associate it to its member entity instances by using their hierarchical relationship in the ontology. The combination of outputs of member entity instances seems straightforward (the module "data combination" in fig 5.5). However, a big question remains: how to pass an order from the discrete controller applied on the entity groups onto some of its member entity instances to get the control done in the physical world? The actions to be taken should be just necessary to respond to the control, more concretely, if turning off one lamp shall be enough, a second lamp should not be turned off.

In order to meet the above requirements, a module called "dispatcher" is located between entity group output and input of member entity instances to send necessary action signal to designated entity instances. This module has the knowledge of the entity group's input/output expressions in terms of states of member entities or some specific value, together with the output value of each member entity and the one of the entity group under control, as shown in fig 5.5.

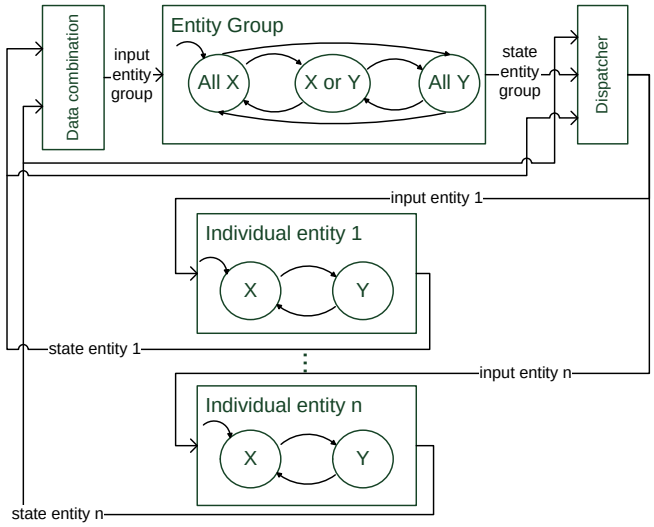


Figure 5.5 – state data combination of member entities and control order dispatching

Algorithm 5 Control dispatching for 3-state general entity group

Require: known state expressions: $All\,X = \neg(\bigvee e.y)$, $All\,Y = \bigwedge e.y$ and $X\,or\,Y = \neg(\bigwedge e.y) \wedge \bigvee e.y$

Require: the result mapping of algorithm 4 for entity group eg and its member entities $\{e_i\}$

if $eg.All\,X\,\&\,\bigvee e.y$ **then** ▷ eg is controlled to go to $All\,X$ state
  for all e_i **do**
    if $e_i.y$ **then** ▷ every member entity in state Y should go to state X
      $e_i.go(X)$;
    end if
  end for
else if $eg.All\,Y\,\&\,\neg\bigwedge e.y$ **then** ▷ eg is controlled to go to $All\,Y$ state
  for all e_i **do**
    if $\neg e_i.y$ **then** ▷ every member entity in state X should go to state Y
      $e_i.go(Y)$;
    end if
  end for
else if $eg.X\,or\,Y$ **then** ▷ entity group is in the intermediate state
  if $\bigwedge e.y$ **then** ▷ All member entities are in state Y
    $\{e_i\}.getOneEntity().go(X)$ ▷ one entity going to X is enough
  else if $\neg\bigvee e.y$ **then** ▷ All member entities are in state X
    $\{e_i\}.getOneEntity().go(Y)$ ▷ one entity going to Y is enough
  end if
end if

The dispatcher takes the uncontrolled member entities state values to calculate the eventual "uncontrolled" result of entity group state to be compared to the controlled one. If they are different, which means the controller has interfered, the dispatcher should send the action signal to those entity instances with a target state to be reached in order to make the "uncontrolled" group state value identical to the controlled one which is the true value of the given expression. We explain this procedure though the 3-state generic entity group for those 2-state entity instances as in 5.3. Algorithm 5 shows the procedure in detail.

Remark 9. The algorithm does not necessarily need to know the exact input of the target member entity that should be actuated. In fact, the specific logic of state transition is encapsulated inside the function or method whose API is provided by the member entity. In the present case, the available API is the function `go(targetState)`. This API is called *service* in OSGi that we will see in chapter 6 about the implementation, □

In the example in figure 5.4, if the entity group EG_A is commanded to state $All\ A2$ which imposes

$$\bigwedge e.A2 = a.A2 \wedge (b.B2 \vee b.B3) \wedge c.C2 \wedge d.D2$$

while the actual states of the member entities $a.A2$, $b.B1$, $c.C2$, $d.D1$ which make the above value false. By applying algorithm 5, an inconsistence of control result and real value is detected, so that actions are sent to those entities which are not in the *correct* state: $b.go(A2)$ and $d.go(A2)$. As state $A2$ has 2 corresponding states in b, it chooses one arbitrarily, e.g. $b.B3$.

5.2 Controller generation using BZR

The controller acting on entity groups is automatically generated using BZR. This generation consists of three steps: first, encode the system model in BZR; second, translate the control objectives into BZR contracts; finally, compile the above codes to get a executable program of the controller with the modeled system. This section describes briefly this process by using the toy example in figure 5.4.

5.2.1 BZR encoding of the system model

In general cases, for the entity group models which are obtained following the very generic template introduced in 5.1.1, they can be translated into BZR code easily as a *node* which takes the form like the follows:

```
node entityGroupModel (v1_and,v1_or,...,vn_and,vn_or,c1,c2:bool)
returns (s1,...,sN)
let
  automaton
    state S1 do
      s1 = true; s2 = false; ...; sN = false;
    until (s2_condition) then S2
      |...
      | (sN_condition) then SN
```

```
    state SN do
      s1 = false; s2 = false; ...; sN = true;
    until (s1_condition) then S1
      |...
      | (sN-1_condition) then SN-1
  end;
tel
```

in which `s1_condition` to `sN_condition` are the entering condition in terms of state combination of member entities (here `v1_and, v1_or,..., v1_and, v1_or`) explained in algorithm 1.

In the example in figure 5.4, the entity group model of `EG_A` is encoded following the above template:

```
node entityGroupModelA (a2_and,a2_or,c1,c2:bool)
returns (all_a1,all_a2,a1_a2)
let
  automaton
    state AllA1 do
      all_a1 = true; all_a2 = false; a1_a2 = false;
    until (not a2_and & a2_or) or c1 & c2 then A1A2
      | a2_and or c1 & c2 then AllA2
    state A1A2 do
      all_a1 = false; all_a2 = false; a1_a2 = true;
    until not a2_or or c1 & c2 then AllA1
      | a2_and or c1 & c2 then AllA2
    state AllA2 do
      all_a1 = false; all_a2 = true; a1_a2 = false;
    until not a2_or or c1 & c2 then AllA1
      | (not a2_and & a2_or) or c1 & c2 then A1A2
  end;
tel
```

The system model used by the DCS process consists of the parallel composition of entity groups and relevant observers. If we develop the current example to include an observer *Obs* of a contextual variable *cv*, its global system behavior model is:

$$S = EG\_A \parallel Obs$$

The BZR encoding of the global system behavior can be obtained by composing all the above models by putting them in parallel denoted by `;`, as shown in the following structured in a node named `globalSys` (we don't describe the model of `Obs` that its presence is only to illustrate the principle of parallel composition to obtain a global system behavior model):

```
node globalSys (a2_and,a2_or,cv:bool)
returns (all_a1,all_a2,a1_a2,scv:bool)
contract
  var...
  assume ...
  enforce ...
  with (c1,c2:bool)
```

```
let
  (all_a1,all_a2,a1_a2) = inlined entityGroupModelA(a2_and,a2_or,c1,c2);
  scv = inlined Obs(cv);
tel
```

The contract part in the above encoding will be detailed in the next section. The 2 controllable variables are declared within the contract, and the keyword `inlined` indicates that the marked node instance are taken into account when DCS is performed.

5.2.2 Control objectives as contract

The contract should formalize the constraints on the global system by Boolean expressions in terms of the output of node instances or the input of the global system. This translation of rules to BZR contract is specific to each rule and each global system. We consider a constraint on the current system that the `EG_A` can never be in `AllA1` while the observer is in the state where `scv=true`, this rule is translated in Boolean expression in terms of states combination:

$$\neg(EG\_A.AllA1 \wedge obs.Scv)$$

The omitted contract part of the above BZR encoding for the global system is then encoded in terms of output of the two instances as follows:

```
contract
  var constraint:bool;
  let
    constraint = not (all_a1 & scv);
  tel
  assume true (*no specific assumptions on the system*)
  enforce constraint
  with (c1,c2:bool)
```

Taking together the system model and the contract, the BZR compiler can synthesize a controller in C or Java code automatically satisfying the defined objectives if it exists.

5.3 Corpus of control rules

We are interested in control applications consisting similar categories of subsystems in one domain, which makes them applicable to different instances of the specific domain. Thus we suppose that it is profitable for end users or the application developers to put these kind of rules in a *common base* shared among themselves, which is inspired by the mechanism of sharing available software libraries. If anyone else needs rules for similar purpose of control, he can look into the base to find rules which may fit, or to have some inspirations for developing his own rules. This common base of rules is called the *corpus of control rules* in the thesis.

Two requirements have been identified for the corpus of rules:

- Rules should be expressed in terms of generic categories or properties corresponding to the nodes of the DAG representing the domain ontology in 4.3.2, to make sure that they are also applicable in other instances of the same domain without further customization.
- Rules should be classified into categories by their purpose of control to facilitate the search of rules and the eventual management of priorities of them in case of conflicts.

5.3.1 Generic rules and their categories

Most of today's control applications are about to manage directly the entities and subsystems one by one. Connected objects are identified either by their serial number (ex. RFID) or by IP address in the local network, thus it is natural to manage them in the same way. It is similar to the traditional industrial control where controlled subjects are wired to control unit one by one: any control logic is expressed naturally in terms of atomic controllable unit of object, like a valve, a robot. It is profitable in industrial implants because the configuration stays the same for a long life cycle and a high level reliability is required. However, such analogy is not suitable for our target scope as its characteristics are so different from those ones from industrial domain. Rules should be changed to follow the differences from one environment instance to another. Any configuration change, which is often especially in the home environments, will lead to a re-edition of related rules, which should not be expected as expertise of the target users. Moreover, we think it is more direct concern for the final users about the environmental properties integrally like the level of light, the temperature, the noise than the details of these concerns like the n-th lamp to turn on to light up the room.

Taking an example in home that we meet in our everyday life:

- for safety, when there is someone in the room, there should be some light.
- for security, close/lock wall opening things (window, door, shutter, etc.).
- for energy efficiency, shut down heating equipment if window open
- for comfort, heating equipment is working when the temperature is lower than 18℃ and some one is present.

We can notice that these rules are expressed in terms of generic categories or properties such as *light*, *wall-opening*, *heating* corresponding to the nodes in a shared ontology that may apply not only to particular instances of entities in this home environment instance, but also in most home instances which share the same generic entity categories. Moreover, even in the same home environment instance, this particularity allows to make a configuration change at individual entity level transparent to the rules and the controller constructed from these rules, because the categories under control stay the same.

It should be noted here that we are only addressing logical control which ensures the reachability and invariance (cf.3.3.2) in a discrete event system. More concretely in this thesis, it is about to forbidden the controlled subsystems to stay in a state, to enforce them

to go to a state, or to choose another state to ensure the next evolutions of the subsystem stay only in permitted states.

5.3.2 Category of rules

When several generic rules are picked up from the corpus to define the control objectives in a given environment instance, they may constrain each other mutually. For example, one rule may require heating the room while another constrains the maximum instantaneous power. In the corpus, rules may be grouped into categories with different priorities. We identify four essential categories of control objectives in the IoT/SE:

- Safety. The rules in this category are for the objective of preventing any harm to people in the environment. For example, light up a room whenever some one is inside to prevent the person from falling in the dark. This category is with the highest priority and cannot be overridden by other categories.
- Security. The rules in this category are for the objective of preventing any harm to properties, material or intellectual, in the environment. This category should have second highest priority following the safety one by default, but could be adjusted by user according to specific need even it is not recommended.
- Energy efficiency. The rules in this category are for the objective of reducing the energy consumption in a given environment by using renewable energy or by a more dynamic and reactive management. It is encouraged and recommended for the users to put this category at a higher priority than the next category *comfort*.
- Comfort. The rules in this category are for the objective of providing high level comfort for the human in the environment. The consequence may be a higher energy consumption. Its priority is lower than the previous energy efficiency category by default, but can be adjusted by end user.

This classification does not include the health and assistant category which a lot of applications address. Some of the rules for personal assistance objectives may be covered by the above categories. For example, medication reminder can be classified in comfort, as well as for adaptation of environment according to recognized activities, warning or alarm triggered by not done routine activities in case of monitoring of elder people can be included in safety, etc.

The priority affected to each category will be used to determine which rule overrides the other in a case where two rules imply contradictory actions on the same target. We will elaborate this aspect later in the next section.

5.3.3 Control rules compatibility

Rules are formulated and contributed to the corpus by different contributors at different moments, so that they are not *supposed* to be all consistent in terms of control result under

the same situation. A very typical example is that a energy efficiency rule tries to keep the indoor temperature between 18 and 20 degree (in winter) while a comfort rule tries to keep it between 22 and 24 degree. A solution to deal with the compatibility problems among the rules is indispensable for a correct system execution and a good acceptance of the whole system. Such coordination problem among rules in a given rule engine has been recognized since long time. Many methods have been proposed since, including the one treating it by a DCS approach by verification and control based on behavior models to avoid static (compilation time) and dynamic (in runtime) problems of redundancy, inconsistency, circularity described in [CDR14]. We will address this problem by a different approach by using the concept of *entity group* which does not represent one entity to be controlled, but a set of entities.

5.3.3.1 Controller by category of rules

Contradictory actions on the instances of models given to the DCS tool to generate a controller would be detected during the compilation time. Regarding to the Heptagon/BZR contracts, one or more inlined nodes, a.k.a. the model instances, should be in more than one state to satisfy the constraints, which would lead to failure of generating such a controller by DCS.

As we are using *entity group* as behavior model to provide to the DCS tool with contracts being expressed directly on them, contradictory behaviors at this level are already verified and eliminated when the controller is generated. If the generation fails, the contracts should be re-verified by someone with the expertise to make the controller generation succeed. When the controller starts to get big with many entries of rules, it would be more and more complicated and challenging for a human being to get constraints expressed in the contract compatible with each other to finalize a compilation. On the other hand, the absence of contradictory actions at the level of behavior models on which the DCS is performed does not mean the absence of contradictory actions at the level of individual member entity instance. In fact, two states of two different entity groups can be totally independent and consistent (ex. one rule requires *light-emitting* group to be in *some-light* state and another rule requires the *noise-making* group in *no-noise* state), while the member entities belonging to both groups could be required in different states (ex. supposing that TV belongs to both light-emitting and noise-making group, it could be asked in *ON* state by the former while in *OFF* state by the latter). However, this situation would not always leads to a conflict at the entity instance level because it will depend on the concrete type, state of each member entity instances in the group. In the above example, if the *light-emitting* group includes other types of member entities other than TV, like lamp, solutions exist to avoid contradictory orders on TV instance while keeping the constraints on entity groups satisfied by ordering lamp instance to be *ON*, while there is no other solution if TV is the only member of *light-emitting* group.

In order to make controller of smaller size and have a method to resolve the conflict problem among different rules, instead of one centralized controller for all the necessary rules, we make a controller per category of rules which work together and if there are contradictory actions coming from different controllers, the most priority ones get executed

knowing that each category of rules has a priority compared to the others. As for the rules belonging to the same category, they are considered as equal in priority that if there is a conflict, the first action on the same entity instance gets executed.

5.3.3.2 Resolution of conflict from different controllers with different priority

We will still use two 3-state entity groups to illustrate the resolution process. Some modifications will be applied in the algorithm 5 to determine the one and correct action signal to be sent to necessary member entities. In the current algorithm, actions are directly sent to the corresponding entities by invoking the `go(target_state)` function. Therefore, to avoid sending inappropriate actions, the target state of one member entity desired by one rule will be temporarily held to be finally verified with all target states on the same entity ordered by others rules and execute the most priority one, which consists of replacing `go(target_state)` by `holdTemporarily(target_state)` in the algorithm. Sometimes there is no desired specific target state for one member entity according to one rule, which implies the member entity can be in any state without any affect on the state of the entity group. If we call the procedure described in algorithm 5 with the above modifications `Procedure preDispatching`, the conflict resolution algorithm is presented in the follows which is executed by the dispatcher module replacing the simple dispatching procedure.

Algorithm 6 Rule conflict resolution

Require: Rule priority

1: $temp$ ▷ $temp$ is the table storing temporarily the target state of every member entity
2: **for all** entity group **do**
3: $PreDispatching()$ ▷ result stored in $temp$
4: **end for**
5: **for all** entity member e_i **do**
6: $finalState \leftarrow highestPriorityTargetState(e_i, temp)$
7: $e_i.go(finalState)$
8: **end for**

Suppose that there are two rules from respectively "safety" and "energy efficiency" category where the former has higher priority than the latter. Entity group A over the category $\mathcal{A}$ including a list of entities $list_A = \{e_i\}_{i=1\cdots n_A}$ is invoked by the safety rule and the entity group B over the category $\mathcal{B}$ including a list of entities $list_B = \{e_i\}_{i=1\cdots n_B}$ is invoked by the energy rule. Some entities belong to both groups and denoted by $list_{common} = \{e_i\}_{i=1\cdots n_c, n_c \leq Min(n_A, n_B)}$. The table to store the temporary target state from each rule is shown in table 5.2.

	e_1	e_2	$\cdots$	e_{n_c}	e_{n_c+1}	$\cdots$	e_{n_B}	e_{n_B+1}	$\cdots$	$e_{n_A+n_B-n_C}$
R_{safety}	N/A	S_1	$\cdots$	S_1	N/A	$\cdots$	N/A	S_1	$\cdots$	S_2, S_3
R_{energy}	S_1	S_1	$\cdots$	S_2	S_1	$\cdots$	S_1	N/A	$\cdots$	N/A
$Result$	S_1	S_1	$\cdots$	S_1	S_1	$\cdots$	S_1	S_1	$\cdots$	S_2, S_3

Table 5.2 – Target states according to different rules from different category

The safety rule has higher priority. In the column e_{n_c}, the two rules define different target state for the entity e_{n_c}, the final target state is obviously S_1 defined by the safety rule. For entity e_1, as the safety rule does not specify a target state, any state would be fine for it, so the result target state is the one defined by energy rule. For e_1, two rules have the same target state, the final state is straightforward. For those entities specific to each group, there would not be any conflict in final target state because they are under only one rule. Taking the last row of the table to be compared to the state value of the entities before control, the `go(target state)` function can be thus called to trigger the necessary and effective actions.

5.3.3.3 Example

We have already mentioned an example in 5.3.3.1 where 2 rules co-exist with potential conflict:

- For safety, when there is someone in the room, there should be some light.
- For comfort/productivity, when someone is working in the room, noise-making things should be turned in "silence" mode.

After a brief observation of the trigger conditions of the two rules, it's obvious that *Working* (in the 2nd rule) is a substate of *Occupied* (in the 1st rule) of the room, so that when the room is in *Working* state, both rules are triggered. Remind here that the two rules belong to two co-existing controllers for different category of rules.

As for the configuration in the room, in this example instance there are 2 lamps, 1 TV and 1 vacuum cleaner following the simplified ontology graph in fig 5.6 (the complete ontology will be given in the case study 7.3).

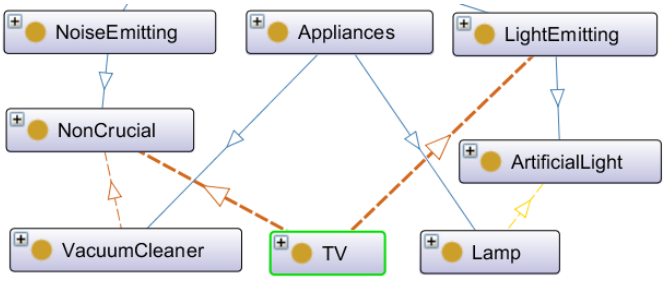


Figure 5.6 – Partial ontology graph for the example

We have seen the VE models for TV, lamp (cf.4.3.1) and light-emitting (cf.4.3.3). Noise-emitting and vacuum cleaner have also 2-state VE model: $\{no{-}noise, noise{-}making\}$ and $\{off, on\}$. Member entity instances of entity group *noise-emitting* and *light-emitting* are quite straightforward to get following the graph as the control is over the scope of one room and all the entity instances are in the single room: $light-emitting : \{lamp1, lamp2, tv\}$, $noise-emitting : \{tv, vacuum\,cleaner\}$ among which the entity instance tv belongs to both groups.

	tv	$lamp1$	$lamp2$	$vacuum\,cleaner$
R_{safety}	N/A	on	N/A	N/A
$R_{comfort}$	off	N/A	N/A	off
$Result$	off	on	N/A	off

Table 5.3 – Target states snapshot when 2 rules execute together

By applying the method introduced in 5.1.2 to determine the state mappings between parent(ancestor)/child(descendant) VE models of categories (not shown in fig 5.6), we can get the list of mapped member entity instance states to each state of the 2 group category VE models:

$$lightEmitting.NoLight = \{lamp1.off, lamp2.off, tv.off\}$$

$$lightEmitting.Lighting = \{lamp1.on, lamp2.on, tv.on\}$$

$$noisemitting.NoNoise = \{tv.off, vacuumCleaner.off\}$$

$$noisemitting.NoiseMaking = \{tv.on, vacuumCleaner.on\}$$

The model for both entity groups is the 3-state model following the general template for a set of 2-state-modeled entity members. We are not detailing their input/output interpretation in terms of logical combination of states of member entity instances here, which can be referred to 5.1.2 where the similar equations have been detailed.

Suppose a scenario where at initial instant, there is no body in the room, all the lights are off but the vacuum cleaner is working itself (a robot). At one moment, a person walks in to start working (event detected by analyzing some sensor data), the room passes to *Working* state and the two rules are executed with the output of the entity groups being:

$$lightEmitting.Lighting = \bigvee e.lighting$$

$$noiseEmitting.NoNoise = \bigwedge e.no{-}noise$$

According to the modified algorithm 5, the first step is to get the target state of each member entity instance assigned by each rule: (table 5.3 and tab:example state compare 2)

The result in table 5.3 shows no contradiction because one of the rules does not need to address the only common entity instance (TV) to get the rule validated (the ON state of entity *lamp1* is enough to make the "**OR**" combination true) , so that the result state of each entity instance is compared to the same state before control to trigger the necessary action on entities. If unfortunately the safety rule chooses to get the common entity instance TV to state *on*, the result will be different as shown in table 5.4. We notice that the two rules gives the contradictory target state to TV. Taking the priority into account, the result target state is the one from the more priority rule. The rule with lower priority is partially executed.

	tv	$lamp1$	$lamp2$	$vacuum\,cleaner$
R_{safety}	**on**	N/A	N/A	N/A
$R_{comfort}$	**off**	N/A	N/A	off
$Result$	**on**	N/A	N/A	off

Table 5.4 – Target states snapshot when 2 rules execute together: another possibility

5.3.3.4 Conclusion of the compatibility resolution

The method we propose here to resolve the compatibility problem is a basic one in the runtime as we suppose the off-line one has already been done by the DCS. Its takes also a "best-effort" approach in a conservative way, which means it does not ever question the decision made by a rule with higher priority before discarding a contradictory action from a rule with lower priority, even there might exist another possibility of solution that satisfy both. In the above example, if the method knew that it could coordinate the decisions of two rules by changing the decision from the safety rule to get tv on to get a lamp on, the decision from the comfort rule could have been completely executed on both TV and vacuum cleaner instances. The reason why this coordination functionality is not integrated is that it turns to re execute the DCS method to find the maximally permissive solution by analyzing all the possible combinations of the states, which we have proposed the concept of "entity group" to get rid of. However, the native non-integration of this feature does not prevent an application-specific configuration given to the dispatcher to tell it how it deal with it in the specific situation, for example, to give a priority to each child category the entity group category for each rule that the entity instances belonging to the most priority category always get order in the first.

Part III

Validation

Chapter 6

Implementation

Contents

This chapter will describe the implementation of the concept presented in the previous two chapters. This implementation develops the prototype implemented in [Hu14] and works together with other modules in lower layers to realize a complete functional loop from the physical world up to the supervisory control block.

6.1 Overall functional architecture

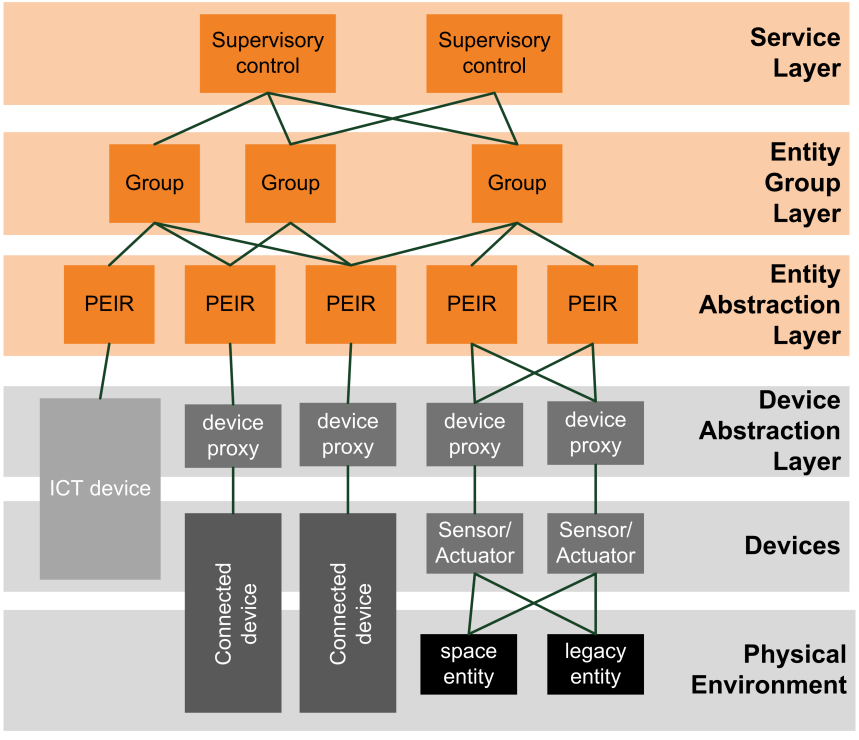


Figure 6.1 – Overall functional architecture of the implementation

We focus on the implementation of the 3 layers colored by orange in figure 6.1 with the support of lower-level layers colored in gray. The two parts make up the physical plane and the proxy instance plane presented in the general framework. The model plane is considered as *knowledge* to the platform which is *static* and *external*, that's why it is not shown in this *functional* architecture.

The platform encompassing the 3 superior layers maintains a 1-to-1 mapped virtual representation for each physical entity as a PEIR (**P**hysical **E**ntity **I**nformation **R**epresentative), an instance for each entity group as well as the controllers generated by the DCS process together with the entity groups on which they apply. The n-to-n mapping between PEIRs and groups are illustrated clearly in the figure. This platform exposes RESTful interfaces to external applications at entity group or entity proxy level, and it accesses to the device abstraction layer to get sensor data or connected/ICT equipment's reading, and to send command to actuators or connected/ICT equipment's settable API. The device abstraction layer is hosted in a home gateway, such as a home set-top box, which may host on the meanwhile the platform itself.

6.2 Ontology implementation

The ontology implementation consists of the description of the domain ontology DAG and the virtual entity model of each node of the DAG in common ontology description language OWL, using the graphical ontology modeling and editing tool Protégé[1]. We will show two pieces of our ontology in the form of Protégé graphical ontology representation, with the help of the plug-in OntoGraf[2]. The first one is a part of the entity categorization in our scope, the second one shows the composition of ontology elements to obtain an automaton.

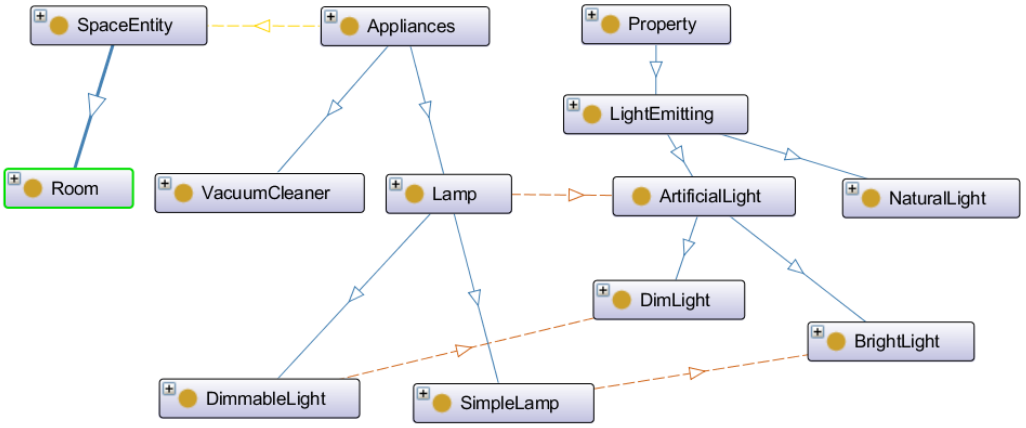


Figure 6.2 – A piece of the implementation of the ontology DAG by Protégé

Figure 6.2 shows a small piece of ontology DAG we implement for one of our study cases. The blue arrows pointing from the top to the bottom denote the relationship "is a" as we explained in 4.3.2 (in an opposite direction as the software interprets the relationship

[1] http://protege.stanford.edu
[2] http://protegewiki.stanford.edu/wiki/OntoGraf

as "has subclass") and the orange arrows denote the "has property" relationship. We can notice that the DAG is divided in three mains branches: appliance, space and property, which correspond respectively to ""thing", "space" and "shared property" that are the three highest-level general classifications in the domain-specific ontology. The yellow arrow provides an additional relationship between the categories "appliance" and "space" which is inherited by all their descendants.

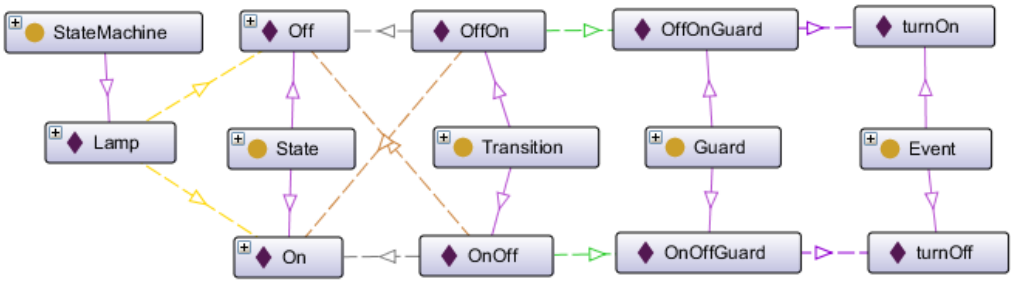


Figure 6.3 – FSM "Lamp" implemented by Protégé

We implement an FSM by instantiating the state machine elements provided in the general state machine ontology proposed by [Dol04] and introduced in 4.3.4, and connecting them by the appropriate relationship also predefined in the same ontology, as shown in 6.3. This "lamp" FSM model composed by the instances of state machine elements is equivalent to the one shown in figure 4.2.

6.3 Implementation on OGSi

OSGi[3] is a component model that enables building component-based Java applications, and running them side-by-side on the same Java Virtual Machine. The OSGi framework is a platform that provides various services to running applications, and enables installing, updating and uninstalling individual applications without restarting the platform. Core features of OSGi are based on an original Java class loader architecture that allows code sharing and isolation between modules called bundles. A bundle contains Java classes that implement zero or more services which are published by the Declarative Services component model to be consumed by other bundles within the same OSGi platform.

The choice of OSGi for the prototype implemented in [Hu14] which serves as a base of the current implementation was the need of creating and reconfiguring the proxy representing each recognized physical entities without stopping and restarting the entire platform to get the reconfigurations applied. This criterion is still valid for current need. Hence we continue the implementation on OSGi by improving the identification and reconfiguration features for the entity proxy level, and adding the entire layers of entity group and supervisory controller as well as some components of supportive functions such as ontology interpretation.

Figure 6.4 shows the modules actually implemented on the OSGi platform (the part surrounded by the blue dotted rectangle) to achieve the full functionality, working together

[3] www.osgi.org

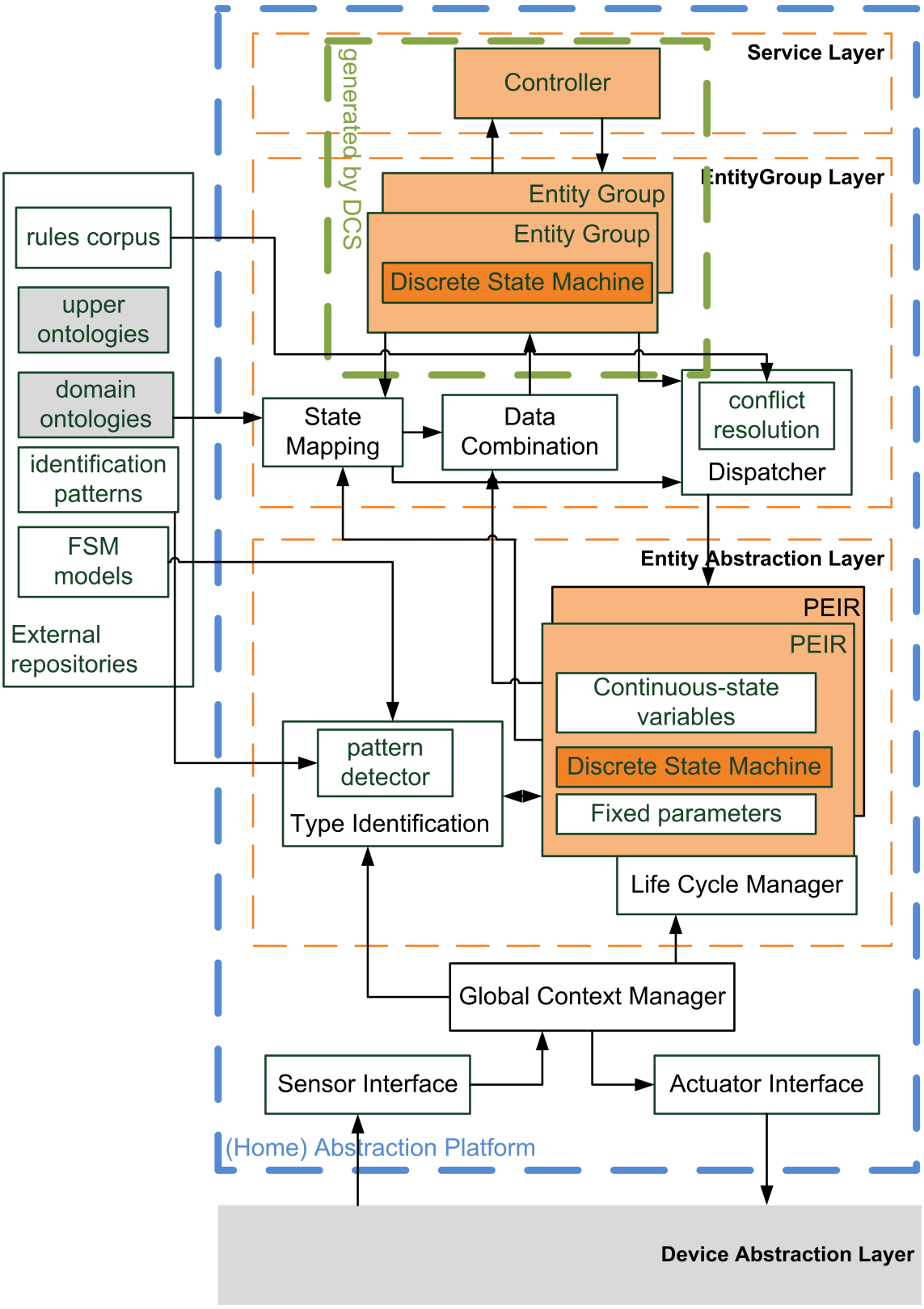


Figure 6.4 – Architecture of the (Home) abstraction platform implemented on OSGi

with external modules via available interfaces. We recognize the orange modules the ones already presented in the functional architecture in figure 6.1 while other white ones ensure the correct performance of the orange blocks by connecting the physical world, interpreting the external knowledge, etc.

Each white block on the platform is implemented as an OSGi bundle providing a service, as well as each PEIR, which can be required and consumed by other bundles. One controller and the entity groups it controls generated together by DCS as an integrated executable function is considered as a bundle. Bundles communicate by intra-OSGi interfaces and the primary bundles hosting the orange modules also provide RESTful interface that can be directly invoked by external applications to get corresponding information on the bundle.

The current implementation is using the Apache Felix implementation of OSGi framework hosted in a Java Virtual Machine 1.6.

6.4 Context simulator

We have developed MiLeSEnS (**Multi Le**vel **S**mart **En**vironment **S**imulator) as a testing ground based on Siafu, an open-source context simulator written in Java[MN06], which provides a GUI and basic and simple physical models such as moving people and physical areas. Figure 6.5 shows an example. These are toy models with no pretense of physical plausibility, yet they may provide enough environmental elements and interactions between actuators and sensors to validate key properties of the models and controllers.

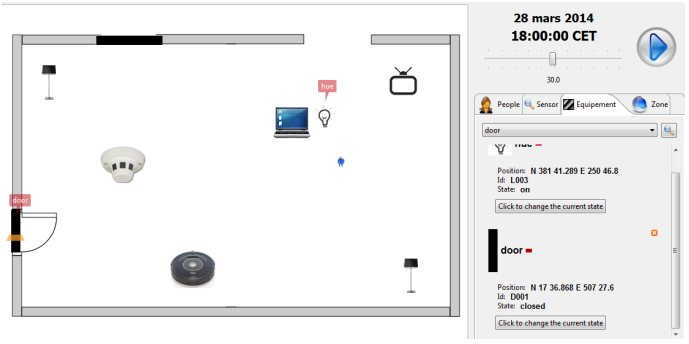


Figure 6.5 – MiLeSEnS GUI example: a home environment interface

The simulated environment includes already the physical environment in the physical plane (c.f. figure 4.1) and the device layer between the two planes as it provides also simulated sensors and actuators. From the point of view of the OSGi-based platform, it is hidden by the device abstraction layer which it sends sensor readings to and takes actuator orders from. A simulation can provide a complete test environment or co-exist in parallel with some pieces of physical equipment monitored and controlled via sensors and actuators thanks to the unification of different sensor/actuator communication protocols (if we consider the simulation output as a "protocol") made by the device abstraction layer.

Chapter 7

Case studies

Contents

7.1 Preliminary: discrete control on generic models for a smart home instance

This preliminary case study consists of applying the DCS technique on the generic entity models for IoT/SE in a home environment with some rules initially on groups in which members share the same property and then interpreted manually in terms of individual entities based on the available ontology. The objective of this study is to validate the generic models without using the complete functionality of the proposed framework.

7.1.1 Case study description

We consider a home environment of one room, with several non-networked appliances: a TV, a lamp, a radiator, a washing machine and an oven, and several infrastructural elements: a window and a door, connected to the control system through the intermediary of networked sensors and actuators, as shown in figure 7.1. Apart from the sensors and

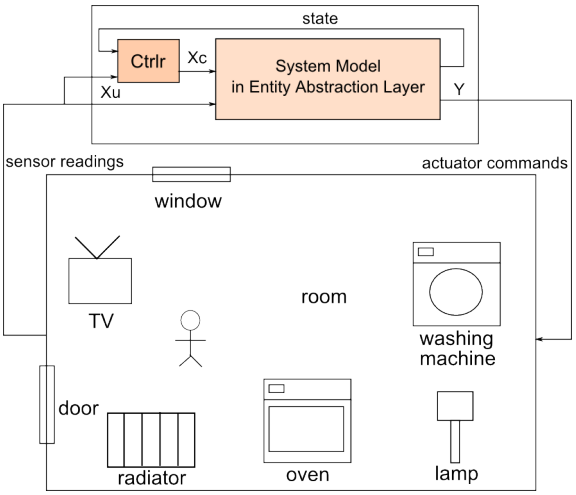


Figure 7.1 – Home environment configuration with control system

actuators connecting "things", there are several infrared sensors detecting the presence and the movement of a living person.

Several control objectives are considered:

1. For safety, at least one light source is on when room is occupied;
2. For security, close window and door when room isn't occupied;
3. For energy efficiency, if any window or door is open, the radiator should not be heating;
4. For energy efficiency, if the room is not occupied, no light is on and the radiator is not heating;

7.1.2 System modeling and discrete controller generation

In this preliminary case study, the concept of "entity group" is not considered as a "system behavior model" to be instantiated and controlled by the controller, but as a "intermediary" to translate rules expressed in terms of generic properties to BZR contract expressed in terms of individual entities. Thus, the generic models for entities in this domain as introduced in 4.3.1 should have controllable variables on their state transitions in order to be controlled directly.

- Door behavior model. It has an initial state *Closed* and a second state *Open*. Input *push* is uncontrollable command from people while the other input c is controllable which will prevent the door from opening or force it to shut, to meet the objectives. Window has the same behavior model. (figure 7.2(a))
- Lamp behavior model. It has an initial state *OFF* and a second state *ON*. Input *turn_on* and *turn_off* are uncontrollable inputs from the switch signal while the

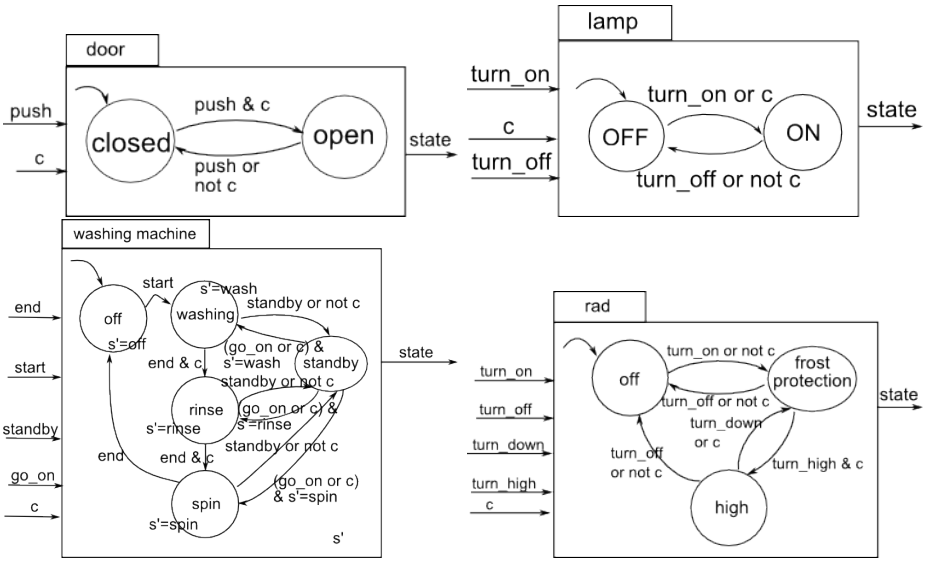


Figure 7.2 – Model with control of (a)Door; (b)Lamp; (c)Washing machine; (d)Radiator

other input c is controllable. TV and oven have the same behavior model. (figure 7.2(b))

- Washing machine behavior model. Washing machine has 2 states without power consumption: *off* and *stand by*, 3 states *washing*, *rinsing* and *spinning* represent the different steps in a washing cycle existing in all modern washing machines. Controllable variable c forces the machine to go to *stand by* states instead of the next working state when necessary. (figure 7.2(c))

- Radiator behavior model. A radiator has state *off* without power consumption and 2 states *frost protection*, *high* with different set temperature hence different electrical power. Controllable variable c forces some transitions and prevent some transitions. (figure 7.2(d))

- Room behavior model. The state transition of a room is uncontrollable as we can not prevent a person from entering or leaving the room. Thus, its behavior model is exactly the same as the one presented in figure 4.3 with 2 states *empty* and *occupied*, and two inputs `presence_on` and `presence_off` which may be the result of several presence sensors.

We formalize the rules for the system in terms of generic properties, eventually shared by more than one individual entities:

1. $room.occupied \Rightarrow lightEmitting.on$;
2. $room.empty \Rightarrow opening.closed$;
3. $opening.open \Rightarrow \neg radiators.high$;
4. $room.empty \Rightarrow lightEmitting.off \wedge \neg radiators.high$

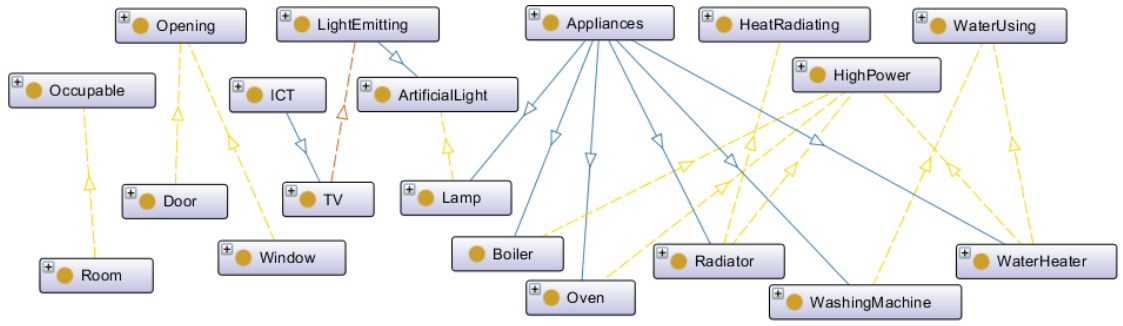


Figure 7.3 – Ontology applied for case study 1 and 2, part taken from a more complete ontology DAG

The domain ontology DAG for this case study is shown in figure 7.3 where the dotted lines mean "has property" and solid blue lines mean "has subclass", according to which we can make groups of individual entities by their shared properties relevant to the rules the system should respect:

- $lightEmitting = \{lamp, tv\}$;
- $opening = \{window, door\}$

Therefore, we can translate the expressions in terms of generic properties used in the rules:

- $lightEmitting.on = lamp.on \vee tv.on$;
- $opening.closed = window.closed \wedge door.closed$;
- $opening.open = window.open \vee door.open$;
- $radiators.high = radiator.high$, in this case, there is only one entity in the group $radiators$;
- $lightEmitting.off = lamp.off \wedge tv.off$

With the above models and rule expressions, the global system is the parallel composition of all the individual entities with control specified by the BZR contract:

```
var safety, security, energy1, energy2:bool;
let
  safety = not roomOccupied or (lamp_on or tv_on);
  security = roomOccupied or (window_closed & door_closed);
  energy1 = not (window_open or door open) or not radiator_high;
  energy2 = roomOccupied or (lamp_off & tv_off) and not radiator_high;
tel
assume true;
enforce safety & security & energy1 & energy2
with (c1,c2,...:bool)
```

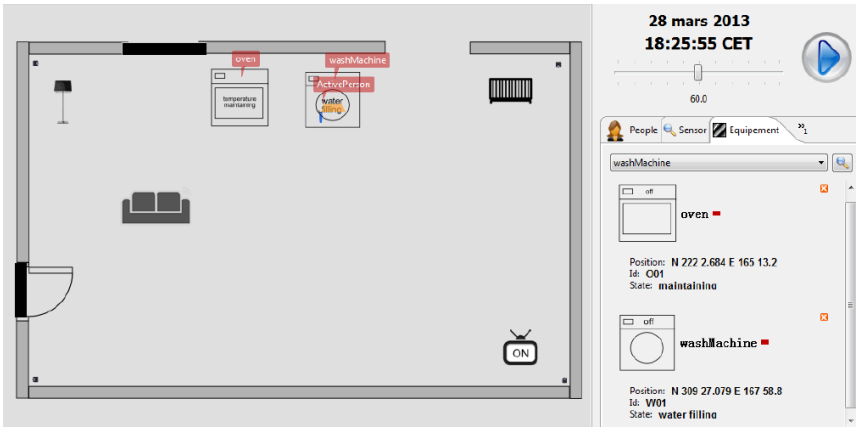


Figure 7.4 – Case study 1: Simulation with synthesized controller

7.1.3 Implementation and simulation

By feeding the BZR program to the BZR compiler, it generates a controller in Java code automatically which has 2 main functions: *step* and *reset*. *reset* initializes the state of the program and *step* executes one reaction where all necessary events are represented by an input of *step* which, after one execution, returns output values of current state of the program.

We programmed a simulation on MiLeSEnS which represents the target environment with all entities and possible system behaviors in absence of control, providing the input for *step* function via available interfaces. Unlike the executions without control where undesired system behaviors are observed, e.g. the radiator is heating with high power while the window is open, the controlled executions respect all the objectives. We can see on the GUI that, as illustrated in Figure 7.4, every time the person does some action or an event occurs, the system never goes to a forbidden state.

7.2 Power control for home: load shedding

Load shedding refers to monitor the energy consumption continuously and disconnects automatically the electric current on some lines in case of overload. It is now a common controlled alternative response to excessive demand in the Grids to avoid a complete black out. It is selective to ensure the continuity of crucial services.

Usually, we identify two main types of load-shedding: *power-based* and *energy-based*. The former one consists of controlling the instant total power to avoid a over voltage while the latter one consists of controlling the total energy consumption in a defined period to avoid a complete run out of energy storage especially in the energy-autonomy cases.

Concerning the individual grid connected environments, load shedding is often used, today, in industrial, large commercial, and utility operations monitoring and shutting down some pre-arranged electric loads when some upper threshold is approached or reached, especially for economic reasons. Though the scope of application is currently in larger

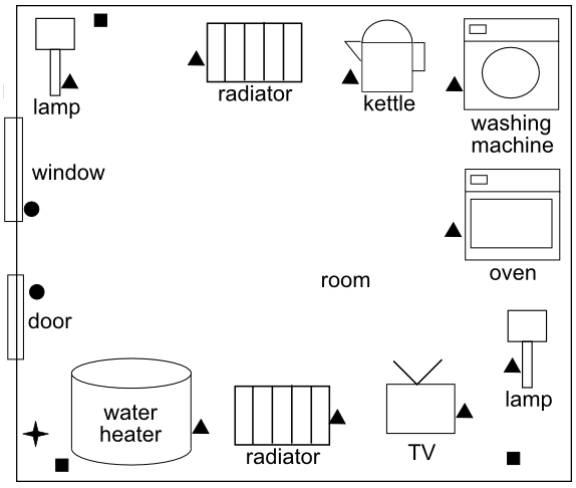


Figure 7.5 – Home instance example

scales of environment, it is certainly applicable in home area by analogy and profitable for the individual end users as well as for the power companies.

The objective of this study is to realize a discrete control on generic environmental properties through the proposed framework. We will describe more in details the procedures introduced in chapter 5 in a general way through the present case study.

7.2.1 Case study description

In the present load shedding scenario, two rules of safety are considered (chosen from the corpus of generic rules):

- When there is someone in the room, there should be some light.
- The instantaneous total power of electrical appliances shall not exceed a threshold set by the grid operator

with two lamp instances, one tv, two radiators, an kettle, an oven, a washing machine in a room, shown in figure 7.5. This home environment instance adopts the ontology DAG already presented in the previous case study in figure 7.3.

7.2.2 System modeling and controller generation

By analyzing the two control rules, the target models directly invoked by the discrete controller are the **room** as observer, the **power observer**, the **light-emitting** entity group and the **high power** entity group. The room observer has only one room, so that it has the same behavior model as the generic room FSM model with 2 states (shown previously in figure 4.3). The power observer monitors the high power appliance entity group (figure 7.6(a)), taking as input the sum of the power of each member entity instance and the threshold value which is given by a `grid observer` transmitting the constraints

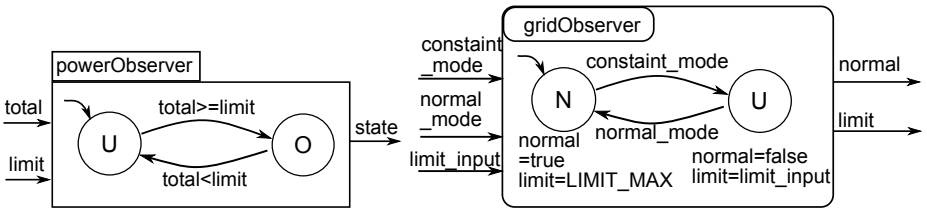


Figure 7.6 – (a)Power observer and (b) grid observer

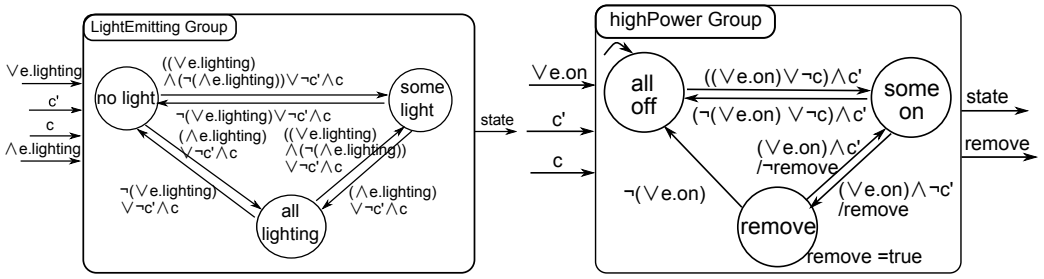


Figure 7.7 – Entity group model of (a)light emitting category; (b)high power appliance

over the entire/partial grid from the electricity supplier. More precisely, the grid observer (figure 7.6(b)) takes the order from the external grid by the binary input `constraint_mode` and `normal_mode` indicating the current situation of the grid: if it is in a normal mode, the threshold of power depends only on the capacity of the individual home, which is a constant `LIMIT_MAX`; if there is constraint outside, the threshold is then defined by the supplier by the input `limit_input`. We can notice that these models are uncontrollable.

As for our target of control, the group of *light-emitting* and *high power*, a FSM model with controllable variables is needed for each of them. The former will be constructed from the generic template, and the latter will be designed specifically for particular objective.

Light-emitting group model is made based on the 2-state group category VE model `no-light/lighting` (shown in fig 4.6). The entity group model is shown in fig 7.7, with 3 states `no light/some lighting/all lighting`, representing respectively the member entities "all in `no-light`", "some in `no-light` and some in `lighting`" and "all in `lighting`", taking as input the logical "OR" and "AND" operation on the state value of members and outputting the actual collective state that the member entities are or should be in. The transitions are controllable which may be inhibited by the Boolean variable `c` or enforced by `c'`, placed on transition labels taking the controller's order. The output state values interpreted in terms of combination of member entity states are:

$$
\begin{aligned}
lightEmittingGroup.noLight &= \bigwedge e.noLight = \neg(\bigvee e.lighting) \\
lightEmittingGroup.someLight &= (\bigvee e.lighting) \wedge (\bigvee e.noLight) \\
&= (\bigvee e.lighting) \wedge \neg(\bigwedge e.lighting) \\
lightEmittingGroup.allLighting &= \bigwedge e.lighting
\end{aligned}
$$

High power group category VE has 2 states `OFF/ON`. Its entity group model (fig-

ure 7.7(b)) is designed to be able to turn off its member entities one by one until the total instantaneous power falls below the threshold, hence a dedicated state `REMOVE` is created in addition compared to the basic model. It takes as input the logical "OR" operation on the state value of *high power* member entities, and outputs the actual collective state that the members are or should be in, as well as the `remove` variable whose true value indicates one *high power* entity should be turned off. When the state is `REMOVE`, the variable `remove` is assigned `true`. When the total power is above the threshold, the entity group should go to the `REMOVE` state, and go back to the normal working state `Some_on` as soon as the threshold is respected. 2 Boolean controllable variables `c` and `c'` are placed on the transition labels to inhibit or enforce the transition to take place. Its output state value can be expressed in terms of member entity states and specific action value:

$$highPowerGroup.allOff = \bigwedge e.off \wedge \neg remove$$

$$highPowerGroup.someOn = \bigvee e.on \wedge \bigvee e.off \wedge \neg remove$$

$$highPowerGroup.remove = \bigvee e.on \wedge \bigvee e.off \wedge remove$$

The global system behavior of the control target system is the parallel composition of the above models which represents all the possible behaviors in the absence of controller:

$$S = lightEmittinggroup \parallel highPowergroup \parallel room \parallel powerObserver \parallel gridObserver$$

The BZR encoding of the global system behavior can be obtained by composing all the above models by putting them in parallel denoted by `;`, as shown in the following structured in a node named `globalSys`:

With the above models (detailed BZR encoding in A.1), observers and entity groups, the global system is their parallel composition with controllable variables specified by the BZR contract:

```
node globalSys (lighting_and,lighting_or,on_or,presence_on,
presence_off,normal_mode:bool;total,limit_input:int)
returns (no_light,all_light,all_off remove,presence,overload,normal:
bool;limit:int)
contract
  var...
  assume ...
  enforce ...
  with (cl1, cl2, ch1, ch2:bool)

let
  (no_light,all_light) = inlined lightEmittingGroup(lighting_and,
  lighting_or, cl1, cl2);
  (all_off,remove) = inlined highPowerGroup(on_or,ch1,ch2);
  presence = inlined room(presence_on,presence_off);
  overload = inlined powerObserver(total,limit);
  (normal,limit) = inlined gridObserver(constraint_mode,normal_mode,
  limit_input);
tel
```

The 2 control objectives can be formalized as Boolean expressions based on the states of the behavior models and observers:

1. $\neg(room.occupied \wedge lightEmittingGroup.noLight)$, exclusivity of the two underlined states
2. $\neg(powerObserver.overload \wedge highPowerGroup.someOn)$, exclusivity of the two underlined states

The omitted contract part of the above BZR encoding for the global system is then encoded as follows:

```
contract
  var exlusiveLight, exclusiveHighPower:bool;
  let
    exlusiveLight = not (presence & no_light);
    exclusiveHighPower = not (overload & (not all_off & not remove));
    (*not all_off & not remove refers to some_on state of high power group*)
  tel
  assume true (*no specific assumptions on the system*)
  enforce exlusiveLight &  exclusiveHighPower
  with (cl1, cl2, ch1, ch2:bool)
```

Taking together the system model and the contract, the BZR compiler can synthesize a controller in C or Java code automatically satisfying the defined objectives if it exists.

7.2.3 Closing the control loop using the platform

Once the models used by DCS are ready, the second step is to establish the connection between entity groups and member entities so that the entity group would represent the actual behavior of the system and the actions would get to the right entity. We apply the method introduced in 5.1.2 over the ontology to get the VE model state mapping tables and instance state mapping which is maintained in the `State Mapping` module. We illustrate then how the `Data Combination` and `Dispatcher` module work through this case study.

Table 7.1 shows, respectively for the two group category VEs, the state mapping of their descendant VE models which are instantiated in the current environment.

According to the above state mapping in 7.1, the two sets of mapped states of member entities respectively to the two states of *light-emitting* VE model are:

$$lightEmitting.NoLight = \{lamp1.off, lamp2.off, tv.off\}$$

$$lightEmitting.Lighting = \{lamp1.on, lamp2.on, tv.on\}$$

And the two sets for *high-power* VE model are:

$$highPower.off = \{kettle.off, oven.off, waterHeater.off, washingMachine.off, washingMachine.standby, radiator1.off, radiator2.off\}$$

$$highPower.on = \{kettle.on, oven.on, waterHeater.on, washingMachine.washing, \\ washingMachine.rinse, washingMachine.spin, \\ radiator1.frostProtection, radiator.high, \\ radiator2.frostProtection, radiator2.high\}$$

High power		
Descendant VE	Parent State	Child(ren) State(s)
Kettle(boiler)	OFF	OFF
	ON	ON
Oven	OFF	OFF
	ON	ON
Water heate	OFF	OFF
	ON	ON
Washing machine	OFF	OFF, stand-by
	ON	Washing, rinse, spin
Radiator	OFF	OFF
	ON	Frost-protection, high

Light emitting		
Descendant VE	Parent State	Child(ren) State(s)
Lamp	No light	OFF
	Lighting	ON
Display	No light	OFF
	Lighting	ON

Table 7.1 – State mapping of ancestor/descendant VE models

With the above sets of mapped states maintained in the `State Mapping` module, the expressions of input/output of the entity groups can be translated:

$$\begin{aligned} \bigwedge lightEmitting.lighting &= \neg(\bigvee lightEmitting.noLight) \\ &= lamp1.on \wedge lamp2.on \wedge tv.on \\ &= \neg(lamp1.off \vee lamp2.off \vee tv.off) \end{aligned}$$

$$\begin{aligned} \bigvee lightEmitting.lighting &= \neg(\bigwedge lightEmitting.noLight) \\ &= lamp1.on \vee lamp2.on \vee tv.on \\ &= \neg(lamp1.off \wedge lamp2.off \wedge tv.off) \end{aligned}$$

$$\begin{aligned} \bigvee highPower.on &= \neg(\bigwedge highPower.off) \\ &= kettle.on \vee oven.on \vee waterHeater.on \vee (washingMachine.washing \\ &\quad \vee washingMachine.rinse \vee washingMachine.spin) \\ &\quad \vee (radiator1.frostProtection \vee radiator1.high) \\ &\quad \vee (radiator2.frostProtection \vee radiator2.high) \\ &= \neg(kettle.off \wedge oven.off \wedge waterHeater.off \end{aligned}$$

$$\wedge (washingMachine.off \vee washingMachine.standby)$$
$$\wedge radiator1.off \wedge radiator2.off)$$

Remark 10. If several states of the same descendant VE model share the same parent state, in the member entity instantiating this model, one and only one state can be active at one time, which explains why they are combined by the "OR" operation to make the parent state active, like $(radiator1.frostProtection \vee radiator1.high) = highPower.on$ in the last equation. □

Replace the right part of the entity group output state expressions by above equations to get the state interpretation in terms of states of present member entities

The total instantaneous power of `high power` group for the power observer is obtained by:

$$\begin{aligned} total =& \sum highPower\_n.power \\ =& kettle.power + oven.power + waterHeater.power \\ &+ washingMachine.power + radiator1.power + radiator2.power \end{aligned}$$

where the value of each $highPower\_n.power$ is provided by electrical sensor associated to the entity instance.

Since this moment, the input values can be calculated by `Data Combination` module without difficulties by applying directly the above equations. The `Dispatcher` module compares the output of `Data Combination` module and the entity group output value, in order to identify the controller's interference, if any, it sends action signal to necessary member entities existing as a PEIR in the runtime.

Remark 11. The entity group `high power` is specifically designed that it has its own order dispatching policy which is not necessarily the same as the policy applied on the entity groups from the generic template. In the present case, the dispatcher sends a `turn_off` signal to one of the member entities which is still on whenever it receives a `remove` from the entity group. □

For example, when the controller enforces the entity group `light-emitting` group to go to state `some light` from state `no light` according to the first rule, the output state value is equivalent to $(\bigvee lightEmitting.lighting) \wedge (\bigvee lightEmitting.noLight)$ while the uncontrolled values would make the left part of the equation, more specifically $\bigvee lightEmitting.lighting$, false. So it turns one entity's state to `lighting`-equivalent state by invoking the entity's `go(lighting-equivalent)` function.

This process of associating member entities to their entity group as well as the correspondence between their input/output values is completely automatic which does not need a case-by-case configuration phase, because all the information needed is in the ontology or in the descriptive file providing necessary knowledge of the entity group models.

7.2.4 Experiments

A graphical tool SIM2CHRO[1] is available in BZR to allow user to perform simulations of the controlled system combined of the generated controller and the global system behavior model by providing the possible sequence of input value. Fig 7.8 shows a scenario of the controlled system implementing the above example by using entity group models.

The first part (marked by **1** with three circles) shows the effects of the safety rule over the light. At initial instant, the detection of person's presence is positive (`presence_on=1`) while the `light-emitting` group is still in `no light` state (`no_light=1`, `all_light=0`) as there is no `light-emitting` entity instance is on (`lighting_or=0`). This situation does not fit the constraint defined in the contract where the controller enforces the `light-emitting` group to go to the state `some light` (`no_light=0`, `all_light=0`). Note that on the graph made by SIM2CHRO, the change of state is only visible at step 2 because the values of these input and output taken to draw the graph are the ones at the beginning of each step, which means the state change taking place at the end of the step would not be seen until the beginning of the next step. The feedback of the actual `light-emitting` entities' state is only effective on step 3 (**lighting_on=1**), which shows that the delay could be non negligible from the moment when the command is sent to the moment when the actions are executed of which the system gets a feedback , but it does not prevent the controller from keeping the constraint respected.

The second part (marked by **2** and **3**) shows the effects of the rule preventing the total electrical power from exceeding a threshold. When the system is in a non-constraint mode where the limit depends only on the capacity of the house (`normal_mode=1`), the total power does not exceed the limit (`(total=56)<(limit=90)`), so that the high power group will go to `some on` state (`all_off=0`, `some_on=1`, `remove=0`) without problem. When constraint from the external grid detected (`constraint_mode=1`), the threshold falls to the value given by the input `limit_input` which is lower than the actual consumption. The power observer thus goes to the `overload` state (`overload=1`), which leads the high power group to go to the remove state (`all_off=0`, `some_on=1`, `remove=1`) until the total power is brought below the threshold (at step 8).

A other implementation of the above system is performed by using directly the individual entity instances as system components to be put in parallel. However these individual entities are instantiated from the more generic level models, which are `highPower` and `lightEmitting`. The control objectives are re-written to take the individual entity instances into account. The objective of this implementation is to compare the cost of the compilation of a controller based on entity group models and the one based on individual entities. The following is the encoding of this program.

The details of the program are given in the A.2. Very intuitively, at encoding level, the program is more complex to manage and error-prone, even the 9 entities are all instances of 2-state models. With such implementation, this controller and the global system behavior model is specific for one configuration of environment instance. Moreover, the size of state space for the synthesis of controller to verify is an exponential function of the number of the

[1] http://www.irisa.fr/vertecs/Logiciels/sigali.html

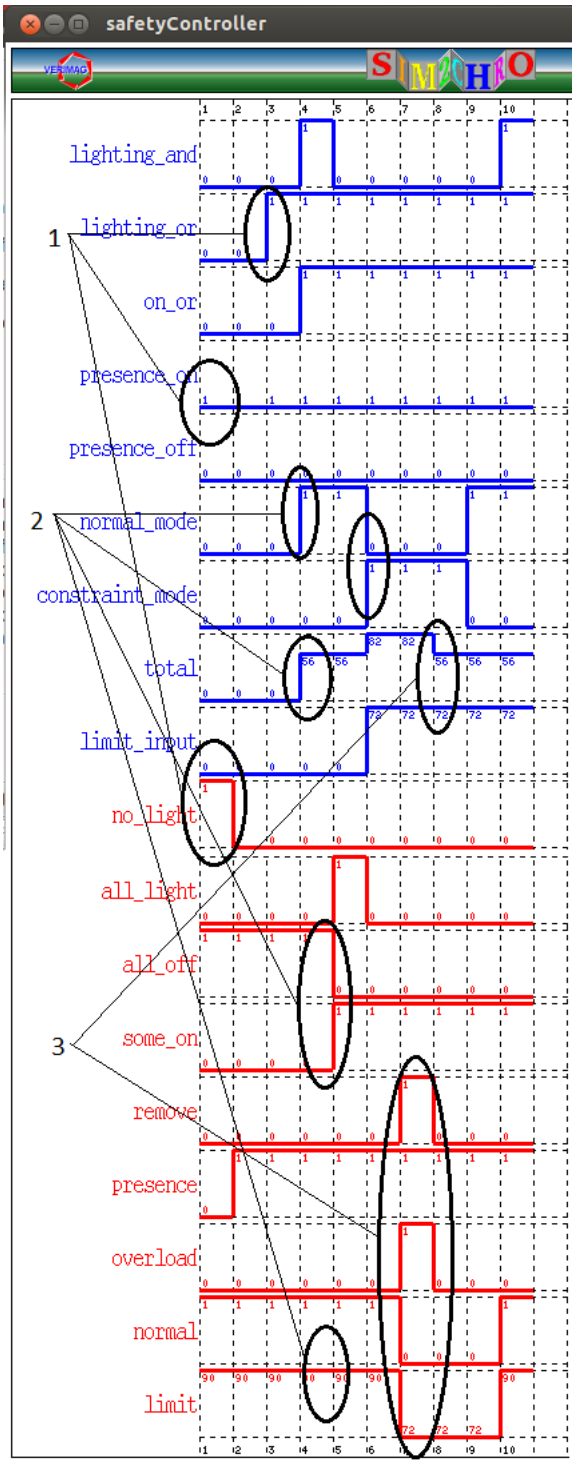


Figure 7.8 – A simulation scenario of safety control

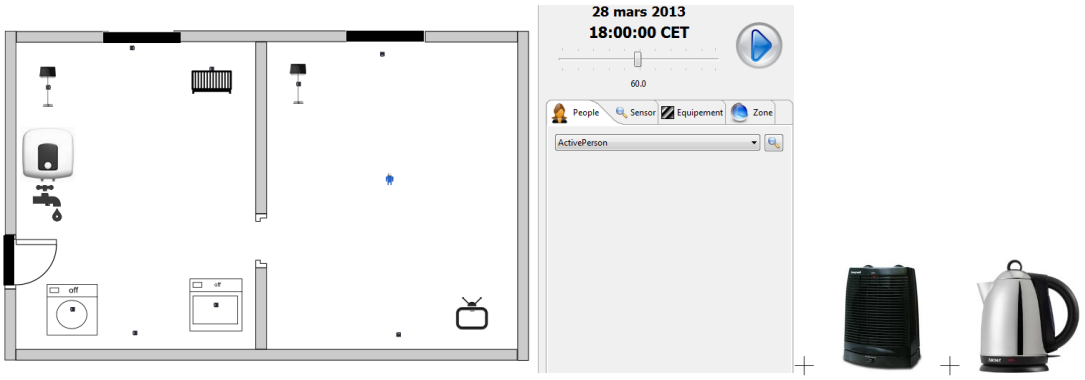


Figure 7.9 – "Mixed" implementation for the case study: kettle and radiator are hardware and others are simulated, including the space entity room

instances put in parallel and their number of states. As a result, the time cost of compilation for such a controller increases. The following table 7.2 illustrates this consequence.

abstraction level	state space size	compilation time (s) (average of 3 executions)
entity group	48	1.0
individual entity component using group category VE	4096	7.0
individual entity component using specific VE	23040	16.0

Table 7.2 – The time costs for DCS operations according to different abstraction level

Another implementation is made as well on individual entity components but this time, they are not all instantiated from more abstract common models, but their own specific "generic" model, like the washing machine has a model with 5 states: *OFF*, *standby*, *washing*, *rinse* and *spin*, which makes the state space of the global system even bigger. We will not give the detailed encoding here (cf. A.3). The result is also put in table 7.2 to be compared.

This case of study is validated by a "mixed" implementation: the complete home configuration is composed by some pieces of home equipment are implemented in the MiLeSEnS simulator and some are real hardware connected by networked sensors and actuators, as shown in figure 7.9. Thanks to the device abstraction layer hiding the physical world from the platform, the whole system makes no difference between the simulated environment and the hardware. The scenarios mentioned above which have validated the discrete control at the entity group level, are re-performed together with simulation and the complete platform. The system constraints are well respected which is visible on the graphical interface of the simulator: when the person moves into a room, the lamp of this room is lighted; when the kettle is turned on making the global power exceed the threshold, we can observe the water heater or the washing machine turned off or put to standby.

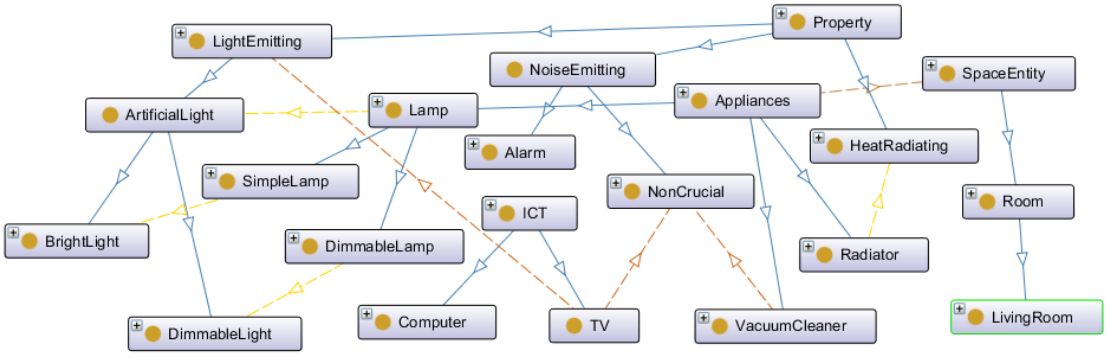


Figure 7.10 – Effective ontology DAG for the home office scenario

7.3 Home office scenario

Thanks to the progress of technology the resent decades, working at home has been enabled by various cooperation tools, such as remote access through VPN, video conference, remote assistance, etc. In order to provide a similar environment as physically in the office to improve productivity, the home environment, at least the room used as working place, should be adapted. In the current scenario, a typical home office context is considered with some common ICT devices for work and other pieces of home equipment. The objective is to provide a quiet, brightly lit environment when the person is identified in work.

7.3.1 Case study description

In the current scenario, 3 points of effect are desired (high level rules):

- The person should not be bothered with non crucial noise (e.g. application notifications, loud appliance noise), but can be bothered by crucial one (e.g. fire alarm);
- Bright light for working ambiance;
- Temperature should be 24°C because the person has no physical activity, and should not be more than 26°C

The home appliances or devices taken into account are: 1 ventilator, a roomba (vacuum cleaning robot), a PC, a TV, a Philips Hue, a simple bulb, and a radiator. The effective ontology (showing only the categories which have one or more direct or descendant instances) for this scenario is shown in figure 7.10.

7.3.2 System modeling and controller generation

The interesting entity groups invoked in the rules are `non-crucial noise`, `crucial noise` or `alarm`, `bright-light`, `dimmable-Light`, `heating`, and necessary observers are `room`, `temperature`.

Some of the relevant entity or category VE models have already been described in other sections, like `simple lamp` (equivalent to `lamp`), `light emitting`, `room`, `radiator`. Vacuum cleaner uses the basic 2-state (`ON/OFF`) model of a normal appliance. The following lists other VEs not yet described in the current DAG, as well as state mapping information registered in the ontology.

- `Dimmable light` and `dimmable lamp`. `Dimmable light` is a child category of the property `lighting emitting`. It has more specific states about the general state `lighting` by distinguishing the light intensity: `dim` and `bright`. This state mapping is shown in figure 4.7. `Dimmable lamp` (figure) also has 3 states corresponding to the 3 states in the property model `dimmable light`.
- `Computer`. The integrated state of a computer would be the full reading of its memory. We simplify it by identifying the current "scene" when it is working: game, multimedia, work, to adapt most of the basic home control. The "scene" is identified either explicitly by the user or implicitly by the activation of some software.
- `Noise`, `alarm` and `non crucial`. The generic model for these 3 categories are identical. It has 2 states: `no noise` and `noise emitting`. The former is mapped by the state `off` of `vacuum cleaner` and `tv`, and the latter is mapped by the state `on`.
- `Living room`. It is child of the room category introduced in the previous case study. It develops the generic occupied state to distinguish a working state for the home office scenario. There could be other activities in the living room to be identified as a state. We are not presenting them for the clarity reason for this case study.
- `Heat radiating`. This property model has 2 states `no heat` and `heating` which correspond respectively to `radiator`'s `off` state and `on` state.

The entity group models are all established applying the generic template on the above group category VEs, as shown in. We do not detail these models as they follow the same principle as the light-emitting group model explained in the previous case study. The global system behavior of the control target system is the parallel composition of the above models which represents all the possible behaviors in the absence of controller:

$$S = brightlightGroup \parallel dimmablelightGroup \parallel non-crucialnoiseGroup \parallel alarmGroup \parallel livingRoom \parallel heatingGroup \parallel temperature$$

Remark 12. The computer is neither a member of an entity group nor an observer directly used in the composition of the global system for the DCS procedure. However, it is useful for the identification of the room's current "scene". From the point of view of the room, the computer is considered as a *virtual sensor* providing some information, as explained in 4.2.3. □

The 3 control objectives can be formalized as Boolean expressions based on the states of the behavior models and observers:

1. $livingroom.working \Rightarrow noncrucialGroup.noNoise$;

2. $livingroom.working \Rightarrow (dimmablelightGroup.somebright \vee dimmablelightGroup.allbright \vee dimmablelightGroup.allMix \vee brightlightGroup.someLight \vee brightlightGroup.allLighting)$

3. $\neg(livingroom.working \wedge temperature.cold \wedge heatingGroup.noHeating)$
   $(livingroom.working \wedge temperature.hot) \Rightarrow heatingGroup.noHeating$

Taking together the system model and the rules transcribed in BZR contract, the BZR compiler can synthesize a controller in C or Java code automatically satisfying the defined objectives if it exists. This controller together with the entity groups are then encapsulated as a bundle in the platform, connecting to individual PEIRs thanks to the supportive functional modules.

7.3.3 Experiments

The validation at the entity group level is done within the simulation tool SIM2CHRO. We can observe that the rules are well respected. We do not show the full screenshot of the SIM2CHRO result because it demonstrates the similar idea as the *load shedding* example. The same reason for not showing the screenshot of the simulation in MiLeSEnS.

The implementation of this case study is also realized in a "mixed" environment: the global context with moving person, alarm system, simple lamps, roomba, is simulated in MiLeSEnS while the dimmable lamp Philips Hue, radiator and TV (replaced by a LCD screen) are hardware equipment. We can observe that if the person is identified as working, the roomba and TV are turned off if they were working, one lamp turned on or Philips Hue turned to bright if it was the only light source in dim mode. And if the temperature is too low or too high, the radiator is actuated.

Part IV

Conclusion

Chapter 8

Conclusion and perspectives

Contents

8.1 Conclusion

The objective of this thesis was to propose a generic modeling framework for monitoring and controlling physical entities in the domain of IoT and Smart Environments in a dynamic and reliable way. We began with identifying the 2 main causes preventing the automatic control applications in this domain from being widely adopted:

- First, mainstream applications of industrail control still rely on fully customized and ad hoc vertically solutions, which makes the cost of a control application too high that of the general public cannot afford it.
- On the other hand, the effectiveness and reliability of current control solutions in the consumer IoT and Smart Environments domain is not satisfying as they adopt the "best-effort" and time-insensitive culture. Non-expected results are sometimes observed due to the lack of formal method.

Our contributions are made to try to fill the huge gap between the above two completely different approaches for control design.

We propose an extension of a framework for a shared generic IoT/SE infrastructure offering high-level interfaces to reduce design effort, and enabling the self-configuration and adaptation of control applications over generic properties of the environment without human interaction. This framework has the general knowledge over the domain which is valid in each target instance of IoT/SE system. Generic FSM models for a type of entity or a shared property are proposed and considered as part of the general knowledge, also called the "domain-specific ontology". An instance of one model can also be considered as an instance of the model of any parent category thanks to the state mapping between

parent and child FSM models. This hierarchical relationship enables the individual physical entities (including all relevant "things", appliances and subsets of space) sharing the same properties, a.k.a. could be modeled by the FSM of the same parent category, being grouped as an integrated virtual entity to be controlled in order to provide a higher level abstraction for control and other applications and better adaptation to lower level configuration changes.

In order to enable applying discrete controller synthesis (DCS) technique at different levels of abstraction and granularity, the groups of entities should also be modeled as FSM. The modeling of an entity group can vary as long as the resulted model represents the desired collective behavior of the given group for one or more control objectives. In absence of specific group model, a general template to help construct a group model is provided within the framework that, according to our point of view, may work for most of the basic control applications in the specific domain. Necessary supporting functional modules are available, such as dynamic state mapping register of the actually present entities and groups, the data combination to provide input data to entity groups and the dispatcher to send command to adequate entities. According to the information registered in the available ontology, they establish the connection between entity groups and individual entity instances and ensure the correct upwards data flow and downwards control command conduction.

A corpus of rules is also proposed in which the rules are expressed on general properties of the environment in order to be useful in most environment instances. Rules are categorized by their priority such as safety and energy efficiency that in case of conflicts of two rules, the one with higher priority is executed.

We implemented the proposed infrastructure by expanding the implementation in [Hu14] and realized several experiments. Models used in these experiments are the generic ones proposed in the framework or designed following the template. For each experiment, a controller is synthesized by applying the DCS technique on the relevant models and a simulation of the target environment, possibly mixed with some pieces of hardware, is performed to validate the scenario. Through these experiments, we showed the feasibility of the approach and the validation of the controller at the level of entity groupings corresponding to rules drawn from the generic corpus.

8.2 Perspectives

8.2.1 Generic and basic control as a "safety guard" service

The trend of the evolution of IoT and Smart Environment platforms is becoming more and more open to host various applications in order to build up an ecosystem satisfying different requirements of different end users. All the applications are obviously developed by numerous developers and like current on line app stores, they should be verified as not malicious before being put on the market by the administrator of the platform. However, the development of each application is completely within the isolated development environment that the developers do not necessarily and cannot predict neither all the interactions

with other co-existing applications, nor all the possible combinations of runtime environment parameters. The random system behavior due to the unpredictable interaction results causes serious problem of system safety and reliability.

We mentioned throughout the thesis that we aimed at a control solution which could realize some basic and generic functionalities as individual applications, like the load shedding control in 7.2 and the home office in 7.3. From the point of view of an open platform, its only duty is to provide a friendly environment for third-party applications and ensure the safety of the global system. Hence, the control solution which is minimum and generic can be considered as a "safety guard" under the application layer which does not substitute any applications and prevents the commands coming from the applications from causing any catastrophic consequence, for example, the "guard" service for safety and energy efficiency in 7.1, or locking the householder inside the house or the basement because of an event resulting from an other application. The "dangerous" combinations of a set of "separatedly safe" actions from co-existing applications are filtered by the discrete control in the "safety guard" layer with higher priority which preempts the command of the applications. This interference from the discrete control occurs only in case of "catastroph" which is specified formally as a system invariant or constraints in the DCS method. Contrast to third-party applications which only have knowledge of their own execution status or relevant environmental parameters, this "safety guard" layer has the full picture of the global system depending on which it can take action before any danger would take place.

8.2.2 Validation beyond home and Application beyond the initial scope

The validation of the proposed framework has been performed with scenarios in the domain of Home. However, the issues that we addressed by this framework through this thesis are common in the domains of home, building and city, as we identified in chapter 1. Hence, this framework is also applicable in all these domains. For example, in the domain of city, the devices and subsystem such as street light and parking are possible to be modeled and controlled in the framework if a corresponding ontology is available.

However, there are some issues which are more present in the domain of city than in the domain of home or building. The most obvious one is the "multiple stakeholders" problem. In a home or building environment, there are rarely more than one "administrator" or manager in the system (the owner of the house for example), while in a city, the subsystems are managed by different operators which have different expertise. How to manage the potential conflicts between these very "closed" subsystems and how to balance their interest (for example, we cannot say in a definitive way that the street lighting system is more/less important than the public parking lighting system to be more/less priority in case of a electricity consumption restriction) should be carefully studied. Secondly, the safety and the time-sensitive aspect is more accentuated in city as a failure of the control system could cause heavier consequence. In order to validate the applicability of the proposed framework, more attention should be payed to the additional issues compared to home, and a simulation should be deployed for the first step of the validation.

We envisage the application of the proposed framework in wider scope of applications, such as classical logistic chain or inventory management, where the infrastructure is cur-

rently designed closed and vertically with one only stakeholder. When the platform in such domain becomes open to host applications from different origins, a comprehensive control application will be meaningless and impossible because there will be always new third-party applications that would have not been taken into account during the control development, therefore a "safety guard" service as mentioned in the previous section will be useful and necessary.

Appendices

Appendix A

BZR encoding of the load shedding case study

A.1 System behavior modeled by groups

```
node powerObserver (exceed:bool)
returns ( overload:bool )
let
  automaton
    state U do
      overload = false;
    until exceed then OVERLOAD
    state OVERLOAD do
      overload = true;
    until not exceed then U
  end;
tel

node gridObserver(constraint_mode,normal_mode:bool;limit_input:int)
returns (normal:bool;limit:int)
let
  automaton
    state N (*normal mode*) do
      normal = true; limit = 90;
    until constraint_mode then U
    state U (*under normal*) do
      normal = false; limit = limit_input;
    until normal_mode then N
  end;
tel

node highPowerGroup (on_or,c1,c2:bool )
returns (off,onn,remove:bool)
let
  automaton
    state AllOFF do
      off = true; onn = false; remove = false;
    until (on_or or not c1) & c2 then SomeON
    state SomeON do
      off = false; onn = true; remove = on_or & not c2;
    until on_or & not c2 then REMOVE
      | (not on_or or not c1) & c2 then AllOFF
    state REMOVE do
      off = false; onn = true; remove = not (on_or & c2);
    until not on_or then AllOFF
      | on_or & c2 then SomeON
```

```
  end;
tel

node lightEmittingGroup(lighting_and,lighting_or,c1,c2:bool)
returns (no_light,all_light:bool)
let
  automaton
    state NoLight do
      no_light = true; all_light = false;
    until (lighting_or or not c1) & c2 then Some_light
    state Some_light do
      all_light = false; no_light = false;
    until (not lighting_or or not c1) & c2 then NoLight
      | (lighting_and or not c1) & c2 then All_lighting
    state All_lighting do
      all_light = true; no_light = false;
    until (not lighting_and or not c1) & c2 then Some_light
  end;
tel

node room (presence_on, presence_off:bool)
returns (presence:bool)
let
  automaton
    state Empty do
      presence = false;
    until presence_on then Occupied
    state Occupied do
      presence = true;
    until presence_off then Empty
  end;
tel

node globalSys (lighting_and, lighting_or, on_or, presence_on, presence_off,
normal_mode,constraint_mode:bool; total, limit_input:int)
returns (no_light, all_light, all_off, some_on, remove, presence, overload, normal:bool;
limit:int)

contract
  var exlusiveLight, exclusiveHighPower:bool;
  let
    exlusiveLight = not (presence & no_light);
    exclusiveHighPower = not (overload & (not all_off & not remove));
  tel
  assume (on_or or not overload)
  enforce exlusiveLight & exclusiveHighPower
  with (cl1, cl2, ch1, ch2:bool)

let
  (no_light, all_light) = inlined lightEmittingGroup(lighting_and, lighting_or,cl1, cl2);
  presence = inlined room (presence_on, presence_off);
  (normal, limit) = inlined gridObserver (constraint_mode, normal_mode,limit_input);
  overload = (total>limit);
  (all_off, some_on, remove) = inlined highPowerGroup(on_or, ch1, ch2);
tel
```

A.2 System behavior as composition of generic individual entities

```
node highPower(turn_on, turn_off, c1, c2:bool)
returns (onn, removed:bool)
let
  automaton
    state OFF do
      onn = false; removed = false;
    until (turn_on or not c1) & c2 then ON
    state ON do
      onn = true; removed = not c1;
    until (turn_off or not c1) & c2 then OFF
  end;
tel

node lightEmitting(turn_on, turn_off, c1, c2:bool)
returns (lighting:bool)
let
  automaton
    state No_light do
      lighting = false;
    until (turn_on or not c1) & c2 then Lighting
    state Lighting do
      lighting = true;
    until (turn_off or not c1) & c2 then No_light
  end;
tel

node powerObserver (exceed:bool)
returns ( overload:bool )
let
  automaton
    state U do
      overload = false;
    until exceed then OVERLOAD
    state OVERLOAD do
      overload = true;
    until not exceed then U
  end;
tel

node gridObserver(constraint_mode,normal_mode:bool; limit_input:int)
returns (normal:bool;limit:int)
let
  automaton
    state N (*normal mode*) do
      normal = true; limit = 90;
    until constraint_mode then U
    state U (*under normal*) do
      normal = false; limit = limit_input;
    until normal_mode then N
  end;
tel

node highPowerGroup (on_or,c1,c2:bool )
returns (off,onn,remove:bool)
```

```
let
  automaton
    state AllOFF do
      off = true; onn = false; remove = false;
    until (on_or or not c1) & c2 then SomeON
    state SomeON do
      off = false; onn = true; remove = on_or & not c2;
    until on_or & not c2 then REMOVE
      | (not on_or or not c1) & c2 then AllOFF
    state REMOVE do
      off = false; onn = true; remove = not (on_or & c2);
    until not on_or then AllOFF
      | on_or & c2 then SomeON
  end;
tel

node lightEmittingGroup(lighting_and,lighting_or,c1,c2:bool)
returns (no_light,all_light:bool)
let
  automaton
    state NoLight do
      no_light = true; all_light = false;
    until (lighting_or or not c1) & c2 then Some_light
    state Some_light do
      all_light = false; no_light = false;
    until (not lighting_or or not c1) & c2 then NoLight
      | (lighting_and or not c1) & c2 then All_lighting
    state All_lighting do
      all_light = true; no_light = false;
    until (not lighting_and or not c1) & c2 then Some_light
  end;
tel

node room (presence_on, presence_off:bool)
returns (presence:bool)
let
  automaton
    state Empty do
      presence = false;
    until presence_on then Occupied
    state Occupied do
      presence = true;
    until presence_off then Empty
  end;
tel

node globalSys (k_turn_on, k_turn_off, o_turn_on, o_turn_off, wh_turn_on,
wh_turn_off, wm_turn_on, wm_turn_off, r1_turn_on, r1_turn_off, r2_turn_on, r2_turn_off,
l1_turn_on, l1_turn_off, l2_turn_on, l2_turn_off, t_turn_on, t_turn_off, presence_on,
presence_off,normal_mode,constraint_mode:bool; total, limit_input:int)
returns (kettleOn,kremoved, ovenOn,oremoved, waterheaterOn,whremoved, washingmachineOn,
wmremoved, radiator1On,r1removed, radiator2On,r2removed, lamp1lighting,lamp2lighting,
tvlighting, presence, overload, normal:bool;limit:int)

contract
  var exlusiveLight, exclusiveHighPower:bool;
  let
    exlusiveLight = not (presence & not (lamp1lighting or lamp2lighting or tvlighting));
```

```
    exclusiveHighPower = not (overload & not (kremoved or oremoved or whremoved
    or wmremoved or r1removed or r2removed));
  tel
  assume ((kettleOn or ovenOn or waterheaterOn or washingmachineOn or radiator1On
  or radiator2On ) or not overload)
  enforce exlusiveLight & exclusiveHighPower
  with (ck1,ck2,co1,co2,cwh1,cwh2,cwm1,cwm2,cr11,cr12,cr21,cr22,cl11,cl12,cl21,
  cl22,ct1,ct2:bool)

let
  (kettleOn,kremoved) = inlined highPower(k_turn_on,k_turn_off,ck1,ck2);
  (ovenOn,oremoved) = inlined highPower(o_turn_on,o_turn_off,co1,co2);
  (waterheaterOn,whremoved) = inlined highPower(wh_turn_on,wh_turn_off,cwh1,cwh2);
  (washingmachineOn,wmremoved) = inlined highPower(wm_turn_on,wm_turn_off,cwm1,cwm2);
  (radiator1On,r1removed) = inlined highPower(r1_turn_on,r1_turn_off,cr11,cr12);
  (radiator2On,r2removed) = inlined highPower(r2_turn_on,r2_turn_off,cr21,cr22);
  lamp1lighting = inlined lightEmitting(l1_turn_on,l1_turn_off,cl11,cl12);
  lamp2lighting = inlined lightEmitting(l2_turn_on,l2_turn_off,cl21,cl22);
  tvlighting = inlined lightEmitting(t_turn_on, t_turn_off, ct1, ct2);
  presence = inlined room (presence_on,presence_off);
  (normal,limit) = inlined gridObserver(constraint_mode,normal_mode,limit_input);
  overload = (total>limit);
tel
```

A.3 System behavior as composition of specific individual entities

```
node highPower(turn_on, turn_off, c1, c2:bool)
returns (onn, removed:bool)
let
  automaton
    state OFF do
      onn = false; removed = false;
    until (turn_on or not c1) & c2 then ON
    state ON do
      onn = true; removed = not c1;
    until (turn_off or not c1) & c2 then OFF
  end;
tel

node lightEmitting(turn_on, turn_off, c1, c2:bool)
returns (lighting:bool)
let
  automaton
    state No_light do
      lighting = false;
    until (turn_on or not c1) & c2 then Lighting
    state Lighting do
      lighting = true;
    until (turn_off or not c1) & c2 then No_light
  end;
tel

node powerObserver (exceed:bool)
returns ( overload:bool )
let
```

```
  automaton
    state U do
      overload = false;
    until exceed then OVERLOAD
    state OVERLOAD do
      overload = true;
    until not exceed then U
  end;
tel

node gridObserver(constraint_mode,normal_mode:bool;limit_input:int)
returns (normal:bool;limit:int)
let
  automaton
    state N (*normal mode*) do
      normal = true; limit = 90;
    until constraint_mode then U
    state U (*under normal*) do
      normal = false; limit = limit_input;
    until normal_mode then N
  end;
tel

node washingMachine(start, e, c1,c2:bool)
returns (off, ws, rs, sp, removed:bool)
var mode_ws,mode_rs,mode_sp:bool;
let
  automaton
    state OFF do
      off = true; ws = false; rs = false; sp = false; removed = false;
      mode_ws = false; mode_rs = false; mode_sp = false;
    until (start or not c1) & c2 then WS
    state Standby do
      off = false; ws = false; rs = false; sp = false; removed = false;
      mode_ws = false; mode_rs = false; mode_sp = false;
    until c2 & mode_ws then WS
      | c2 & mode_rs then RS
      | c2 & mode_sp then SP
    state WS do
      off = false; ws = true; rs = false; sp = false; removed = not c2;
      mode_ws = true; mode_rs = false; mode_sp = false;
    until not c2 then Standby
      | e & c2 then RS
    state RS do
      off = false; ws = false; rs = true; sp = false; removed = not c2;
      mode_ws = false; mode_rs = true; mode_sp = false;
    until not c2 then Standby
      | e & c2 then SP
    state SP do
      off = false; ws = false; rs = false; sp = true; removed = not c2;
      mode_ws = false; mode_rs = false; mode_sp = true;
    until not c2 then Standby
      | e then OFF
  end;
tel

node radiator (up1, down1, up2, down2,(*input events*)
c1,c2:bool (*control events, c1 c2 never all true*))
```

```
returns (fp,hi,removed:bool)
let
  automaton
    state OFF do
      fp = false; hi = false; removed = false;
    until (up1 or not c1) & c2 then FP
    state FP do
      fp = true; hi = false; removed = not c1;
    until down1 or not c1 then OFF
      | up2 & c2 then HI
    state HI do
      fp = false; hi = true; removed = not c1;
    until down1 or not c1 then OFF
      | down2 or not c2 then FP
  end;
tel

node room (presence_on, presence_off:bool)
returns (presence:bool)
let
  automaton
    state Empty do
      presence = false;
    until presence_on then Occupied
    state Occupied do
      presence = true;
    until presence_off then Empty
  end;
tel

node globalSys (k_turn_on,k_turn_off,o_turn_on,o_turn_off,wh_turn_on,wh_turn_off,
wm_turn_on,wm_turn_off,r1_turn_on,r1_turn_off,r1_turn_up,r1_turn_down,r2_turn_on,
r2_turn_off,r2_turn_up,r2_turn_down,l1_turn_on,l1_turn_off,l2_turn_on,l2_turn_off,
t_turn_on,t_turn_off,presence_on,presence_off,normal_mode,constraint_mode:bool;
total, limit_input:int)
returns (kettleOn,kremoved,ovenOn,oremoved,waterheaterOn,whremoved,wmOff,wmws,
wmrs,wmsp,wmremoved,r1fp,r1hi,r1removed,r2fp,r2hi,r2removed,lamp1lighting,
lamp2lighting,tvlighting, presence, overload, normal:bool;limit:int)

contract
  var exlusiveLight, exclusiveHighPower:bool;
  let
    exlusiveLight = not(presence & not(lamp1lighting & lamp2lighting & tvlighting));
    exclusiveHighPower = not(overload & not(kremoved or oremoved or whremoved or
    wmremoved or r1removed or r2removed));
  tel
  assume ((kettleOn or ovenOn or waterheaterOn or wmws or wmrs or wmsp or r1fp or
  r1hi or r2fp or r2hi ) or not overload)
  enforce exlusiveLight & exclusiveHighPower
  with (ck1,ck2,co1,co2,cwh1,cwh2,cwm1,cwm2,cr11,cr12,cr21,cr22,cl11,cl12,cl21,
  cl22,ct1,ct2:bool)

let
  (kettleOn,kremoved) = inlined highPower(k_turn_on,k_turn_off,ck1,ck2);
  (ovenOn,oremoved) = inlined highPower(o_turn_on,o_turn_off,co1,co2);
  (waterheaterOn,whremoved) = inlined highPower(wh_turn_on,wh_turn_off,cwh1,cwh2);
  (wmOff,wmws,wmrs,wmsp,wmremoved) = inlined washingMachine(wm_turn_on,wm_turn_off,
  cwm1,cwm2);
```

```
  (r1fp,r1hi,r1removed) = inlined radiator(r1_turn_on,r1_turn_off,r1_turn_up,
  r1_turn_down,cr11,cr12);
  (r2fp,r2hi,r2removed) = inlined radiator(r2_turn_on,r2_turn_off,r2_turn_up,
  r2_turn_down,cr21,cr22);
  lamp1lighting = inlined lightEmitting(l1_turn_on,l1_turn_off,cl11,cl12);
  lamp2lighting = inlined lightEmitting(l2_turn_on,l2_turn_off,cl21,cl22);
  tvlighting = inlined lightEmitting(t_turn_on,t_turn_off,ct1,ct2);
  presence = inlined room (presence_on,presence_off);
  (normal,limit) = inlined gridObserver (constraint_mode,normal_mode,limit_input);
  overload = (total>limit);
tel
```

List of Publications

[Pri+13] Gilles Privat, Mengxuan Zhao, Eric Rutten, and Hassane Alla. "Configuration automatique du contrôle discret d'entités physiques dans un système de supervision et de contrôle". Pat. FR 1361235. Nov. 2013.

[PZL14] Gilles Privat, Mengxuan Zhao, and Laurent Lemke. "Towards a Shared Software Infrastructure for Smart Homes, Smart Buildings and Smart Cities". In: *1st International Workshop on Emerging Trends in the Engineering of Cyber-Physical Systems, part of CPSWEEK*. (Berlin). Apr. 2014.

[Zha+13a] Mengxuan Zhao, Gilles Privat, Eric Rutten, and Hassane Alla. "Discrete Control for the Internet of Things and Smart Environments". In: *Presented as part of the 8th International Workshop on Feedback Computing*. San Jose, CA: USENIX, 2013.

[Zha+13b] Mengxuan Zhao, Gilles Privat, Eric Rutten, and Hassane Alla. "Modèles génériques applicables à la synthèse de contrôleurs discrets pour l'internet des objets". In: *Journal Européen des Systèmes Automatisés* 47.1-3 (2013), pp. 211–225.

[Zha+14] Mengxuan Zhao, Gilles Privat, Eric Rutten, and Hassane Alla. "Discrete Control for Smart Environments Through a Generic Finite-State-Models-Based Infrastructure". English. In: *Ambient Intelligence*. Ed. by Emile Aarts et al. Lecture Notes in Computer Science 8850. Springer International Publishing, 2014, pp. 174–190.

Bibliography

[Aky+02] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. "Wireless sensor networks: a survey". In: *Computer networks* 38.4 (2002), pp. 393–422 (cit. on p. 14).

[Ala+14] M. Ben Alaya et al. "OM2M: Extensible ETSI-compliant {M2M} Service Platform with Self-configuration Capability". In: *Procedia Computer Science* 32.0 (2014). The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), pp. 1079 –1086 (cit. on p. 10).

[An+13] Xin An et al. "Autonomic Management of Dynamically Partially Reconfigurable FPGA Architectures Using Discrete Control." In: *ICAC*. 2013, pp. 59–63 (cit. on p. 36).

[Ban+11] Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. "Role of middleware for internet of things: A study". In: *International Journal of Computer Science and Engineering Survey* 2.3 (2011), pp. 94–105 (cit. on p. 16).

[Bar+12] Payam Barnaghi, Wei Wang, Cory Henson, and Kerry Taylor. "Semantics for the Internet of Things: early progress and back to the future". In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 8.1 (2012), pp. 1–21 (cit. on p. 18).

[Bat07] Michael Batty. *Cities and Complexity: Understanding Cities with Cellular Automata, Agent-Based Models, and Fractals*. The MIT Press, 2007 (cit. on p. 28).

[BC08] Dario Bonino and Fulvio Corno. *Dogont-ontology modeling for intelligent domotic environments*. Springer, 2008 (cit. on p. 20).

[BG92] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: Design, semantics, implementation". In: *Science of computer programming* 19.2 (1992), pp. 87–152 (cit. on pp. 31, 32).

[CAS] EU CASAGRAS. *FP7 Project, RFID and the inclusive model for the Internet of Things*. Tech. rep. Technical report, 2012. www. grifs-project. eu. 1 (cit. on p. 13).

[Cas06] F. Castiglione. "Agent based modeling". In: *Scholarpedia* 1.10 (2006), p. 1562 (cit. on p. 28).

[CDR14] Julio Cano, Gwenaël Delaval, and Eric Rutten. "Coordination of ECA Rules by Verification and Control". In: *Coordination Models and Languages*. Springer. 2014, pp. 33–48 (cit. on pp. 37, 77).

[Cha96] ANDR Charles. "Representation and analysis of reactive behaviors: A synchronous approach". In: *Computational Engineering in Systems Applications, CESA*. Vol. 96. 1996, pp. 19–29 (cit. on p. 32).

[CHC89] William R Cook, Walter Hill, and Peter S Canning. "Inheritance is not subtyping". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 125–135 (cit. on p. 28).

[Cor+02] Elisabetta Cortese, Filippo Quarta, Giosue Vitaglione, and P Vrba. "Scalability and performance of jade message transport system". In: *AAMAS Workshop on AgentCities, Bologna*. Vol. 16. 2002 (cit. on p. 28).

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "A conservative extension of synchronous data-flow with state machines". In: *Proceedings of the 5th ACM international conference on Embedded software*. ACM. 2005, pp. 173–182 (cit. on p. 32).

[Del+14] Gwenaël Delaval, Soguy Mak-Karé Gueye, Eric Rutten, and Noël De Palma. "Modular coordination of multiple autonomic managers". In: *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*. ACM. 2014, pp. 3–12 (cit. on p. 36).

[Del14] José C Delgado. "Improving Data and Service Interoperability with Structure, Compliance, Conformance and Context Awareness". In: *Big Data and Internet of Things: A Roadmap for Smart Environments*. Springer, 2014, pp. 35–66 (cit. on p. 17).

[Dol04] Peter Dolog. "Model-Driven Navigation Design for Semantic Web Applications with the UML-Guide." In: *ICWE Workshops*. 2004, pp. 75–86 (cit. on pp. 50, 87).

[DR10] Gwenaël Delaval and Eric Rutten. "Reactive model-based control of reconfiguration in the fractal component-based model". In: *Component-Based Software Engineering*. Springer, 2010, pp. 93–112 (cit. on p. 36).

[Fen+13] Anna Fensel et al. "Sesame-s: Semantic smart home system for energy efficiency". In: *Informatik-Spektrum* 36.1 (2013), pp. 46–57 (cit. on p. 18).

[Fie00] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000 (cit. on p. 16).

[For+13] Giancarlo Fortino et al. "An agent-based middleware for cooperating smart objects". In: *Highlights on Practical Applications of Agents and Multi-Agent Systems*. Springer, 2013, pp. 387–398 (cit. on p. 28).

[For82] Charles L Forgy. "Rete: A fast algorithm for the many pattern/many object pattern match problem". In: *Artificial intelligence* 19.1 (1982), pp. 17–37 (cit. on p. 37).

[GBB13] Sébastien Guillet, Bruno Bouchard, and Abdenour Bouzouane. "Correct by construction security approach to design fault tolerant smart homes for disabled people". In: *Procedia Computer Science* 21 (2013), pp. 257–264 (cit. on p. 36).

[GNP13] Marco Grassi, Michele Nucci, and Francesco Piazza. "Ontologies for Smart Homes and Energy Management: an Implementation-driven Survey". In: *Proceedings of the IEEE Workshop on Modeling and Simulation of Cyber-Physical Energy Systems 2013, May 20, 2013, Berkeley, CA*. 2013 (cit. on p. 20).

[Gru93] Thomas R Gruber. "A translation approach to portable ontology specifications". In: *Knowledge acquisition* 5.2 (1993), pp. 199–220 (cit. on pp. 18, 19).

[Gua98] Nicola Guarino. *Formal ontology in information systems: Proceedings of the first international conference (FOIS'98), June 6-8, Trento, Italy*. Vol. 46. IOS press, 1998 (cit. on p. 19).

[Gur+08] Levent Gurgen et al. "SStreaMWare: a service oriented middleware for heterogeneous sensor data management". In: *Proceedings of the 5th international conference on Pervasive services*. ACM. 2008, pp. 121–130 (cit. on p. 15).

[Gyr13] Amelie Gyrard. "A machine-to-machine architecture to merge semantic sensor measurements". In: *Proceedings of the 22nd international conference on World Wide Web companion*. International World Wide Web Conferences Steering Committee. 2013, pp. 371–376 (cit. on p. 20).

[Hal+91] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320 (cit. on p. 31).

[Har+90] David Harel et al. "Statemate: A working environment for the development of complex reactive systems". In: *Software Engineering, IEEE Transactions on* 16.4 (1990), pp. 403–414 (cit. on p. 30).

[Har07] David Harel. "Statecharts in the making: a personal account". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM. 2007, pp. 5–1 (cit. on p. 30).

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274 (cit. on pp. 30, 52).

[HG96] David Harel and Eran Gery. "Executable object modeling with statecharts". In: *Proceedings of the 18th international conference on Software engineering*. IEEE Computer Society. 1996, pp. 246–257 (cit. on p. 31).

[Hu14] Zheng Hu. "Self-configuration, Monitoring and Control of Physical Entities via Sensor and Actuator Networks". Université de Lyon, 2014 (cit. on pp. 13, 43, 85, 87, 112).

[LeG+91] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. "Programming real-time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1321–1336 (cit. on p. 31).

[LF09] Ching-Hu Lu and Li-Chen Fu. "Robust location-aware activity recognition using wireless sensor network in an attentive home". In: *Automation Science and Engineering, IEEE Transactions on* 6.4 (2009), pp. 598–609 (cit. on p. 15).

[LS11] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia, 2011 (cit. on p. 29).

[Mar+00] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic. "Synthesis of discrete-event controllers based on the signal environment". In: *Discrete Event Dynamic Systems* 10.4 (2000), pp. 325–346 (cit. on p. 34).

[Mar+13] José-Fernán Martínez, Jesús Rodríguez-Molina, Pedro Castillejo, and Rubén De Diego. "Middleware architectures for the smart grid: survey and challenges in the foreseeable future". In: *Energies* 6.7 (2013), pp. 3593–3621 (cit. on p. 16).

[Mar91] F Maraninchi. "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems". In: *In IEEE Workshop on Visual Languages*. 1991 (cit. on p. 30).

[MN06] Miquel Martin and Petteri Nurmi. "A generic large scale simulator for ubiquitous computing". In: *Mobile and Ubiquitous Systems: Networking & Services, 2006 Third Annual International Conference on*. IEEE. 2006, pp. 1–3 (cit. on p. 89).

[MR01] Florence Maraninchi and Yann Rémond. "Argos: an automaton-based synchronous language". In: *Computer languages* 27.1 (2001), pp. 61–92 (cit. on pp. 31, 32).

[MR03] Florence Maraninchi and Yann Rémond. "Mode-automata: a new domain-specific construct for the development of safe critical systems". In: *Science of Computer Programming* 46.3 (2003), pp. 219–254 (cit. on p. 32).

[NM+01] Natalya F Noy, Deborah L McGuinness, et al. *Ontology development 101: A guide to creating your first ontology*. 2001 (cit. on p. 19).

[Ove07] Hagen Overdick. "The resource-oriented architecture". In: *Services, 2007 IEEE Congress on*. IEEE. 2007, pp. 340–347 (cit. on p. 16).

[Per+12] Camille Persson, Gauthier Picard, Fano Ramparany, and Olivier Boissier. "A multi-agent based governance of machine-to-machine systems". In: *Multi-Agent Systems*. Springer, 2012, pp. 205–220 (cit. on p. 28).

[Pri12] Gilles Privat. "Phenotropic and stigmergic webs: the new reach of networks". English. In: *Universal Access in the Information Society* 11.3 (2012), pp. 323–335 (cit. on p. 13).

[Rei+11] Christian Reinisch, Mario J. Kofler, Félix Iglesias, and Wolfgang Kastner. "ThinkHome Energy Efficiency in Future Smart Homes". In: *EURASIP J. Embedded Syst.* 2011 (Jan. 2011), 1:1–1:18 (cit. on p. 20).

[RW87] Peter J Ramadge and W Murray Wonham. "Supervisory control of a class of discrete event processes". In: *SIAM journal on control and optimization* 25.1 (1987), pp. 206–230 (cit. on p. 34).

[RWE13] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Incorporated, 2013 (cit. on p. 48).

[SB11] Zach Shelby and Carsten Bormann. *6LoWPAN: The wireless embedded Internet*. Vol. 43. John Wiley & Sons, 2011 (cit. on p. 14).

[SCM10] Zhexuan Song, Alvaro A Cárdenas, and Ryusuke Masuoka. "Semantic middleware for the Internet of Things". In: *Internet of Things (IOT), 2010*. IEEE. 2010, pp. 1–8 (cit. on p. 18).

[Sel03] Bran Selic. "The pragmatics of model-driven development". In: *IEEE software* 20.5 (2003), pp. 19–25 (cit. on p. 27).

[SM00] Miro Samek and Paul Y Montgomery. "State oriented programming". In: *Embedded Systems Programming* 13.8 (2000), pp. 22–43 (cit. on p. 52).

[Spi+09] Patrik Spiess et al. "SOA-based integration of the internet of things in enterprise services". In: *Web Services, 2009. ICWS 2009. IEEE International Conference on*. IEEE. 2009, pp. 968–975 (cit. on p. 16).

[Sta+12] Thanos G Stavropoulos, Dimitris Vrakas, Danai Vlachava, and Nick Bassiliades. "Bonsai: a smart building ontology for ambient intelligence". In: *Proceedings of the 2nd International Conference on Web Intelligence, Mining and Semantics*. ACM. 2012, p. 30 (cit. on p. 20).

[Tap+06] Emmanuel Munguia Tapia, Stephen S Intille, Louis Lopez, and Kent Larson. "The design of a portable kit of wireless sensors for naturalistic data collection". In: *Pervasive Computing*. Springer, 2006, pp. 117–134 (cit. on p. 15).

[Tei+11] Thiago Teixeira, Sara Hachem, Valérie Issarny, and Nikolaos Georgantas. "Service oriented middleware for the internet of things: a perspective". In: *Towards a Service-Based Internet*. Springer, 2011, pp. 220–229 (cit. on p. 16).

[TIL04] Emmanuel Munguia Tapia, Stephen S Intille, and Kent Larson. *Activity recognition in the home using simple and ubiquitous sensors*. Springer, 2004 (cit. on p. 15).

[VM11] Bruno Valente and Francisco Martins. "A middleware framework for the Internet of Things". In: *AFIN 2011, The Third International Conference on Advances in Future Internet*. 2011, pp. 139–144 (cit. on p. 16).

[Yan00] Mihalis Yannakakis. "Hierarchical state machines". In: *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*. Springer, 2000, pp. 315–330 (cit. on p. 30).