A Semantics-Rich Information Technology Architecture for Smart Buildings

Buildings **2014**, *4*, 880-910; doi:10.3390/buildings4040880

OPEN ACCESS

buildings

ISSN 2075-5309

www.mdpi.com/journal/buildings

Article

A Semantics-Rich Information Technology Architecture for Smart Buildings

Dario Bonino, Fulvio Corno * and Luigi De Russis

Department of Control and Computer Engineering, Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Torino, Italy; E-Mails: dario.bonino@polito.it (D.B.); luigi.derussis@polito.it (L.D.R.)

* Author to whom correspondence should be addressed; E-Mail: fulvio.corno@polito.it; Tel.: +39-011-0907053.

External Editor: Gyu Myoung Lee

Received: 25 July 2014; in revised form: 21 October 2014 / Accepted: 21 October 2014 / Published: 4 November 2014

Abstract: The design of smart homes, buildings and environments currently suffers from a low maturity of available methodologies and tools. Technologies, devices and protocols strongly bias the design process towards vertical integration, and more flexible solutions based on separation of design concerns are seldom applied. As a result, the current landscape of smart environments is mostly populated by defectively designed solutions where application requirements (e.g., end-user functionality) are too often mixed and intertwined with technical requirements (e.g., managing the network of devices). A mature and effective design process must, instead, rely on a clear separation between the application layer and the underlying enabling technologies, to enable effective design reuse. The role of smart gateways is to enable this separation of concerns and to provide an abstracted view of available automation technology to higher software layers. This paper presents a blueprint for the information technology (IT) architecture of smart buildings that builds on top of established software engineering practices, such as model-driven development and semantic representation, and that avoids many pitfalls inherent in legacy approaches. The paper will also present a representative use case where the approach has been applied and the corresponding modeling and software tools.

Keywords: smart buildings; smart gateway; semantic web; interoperability; OSGi; model-driven

---

1. Introduction

The construction of smart buildings and, more generally, the adoption of advanced monitoring, control and automation technologies in building management are increasingly important, as these allow one to meet ever-increasing energy efficiency goals, to match sophisticated personalization in user comfort, to enhance security and, more generally, to achieve higher levels of efficiency.

The adoption of building automation systems (BAS), building management systems (BMS) or energy management systems (EMS), able to integrate and manage several kinds of automation technologies, is now a mandatory task, and the "intelligent" subsystems in a building are constantly increasing in number and complexity. However, the design methodologies have seldom adapted to the new requirements, and the intelligent nature of new building automation plants is not recognized. In other words, buildings host several intelligent plants, which fulfill different design goals (e.g., energy distribution, air treatment and conditioning, surveillance, *etc*.), but do not share information nor common strategies. This is partly due to the sheer variety of available technologies, which have evolved independently and in competition with each other, and to the management and subcontracting processes, which naturally tend to partition the work into smaller independent units.

The overall architecture of BMS and EMS systems is described in detail in the comprehensive handbook edited by Capehart and Middelkoop [1], which covers a quite recent overview over existing architectures and focuses in particular on the type of applications and on the data elaborations that enable energy management and savings. The handbook mentions several times the issue of interoperability, although in a limited fashion: they mainly refer to the integration between different information systems (and at this level, there is very little dependence on the physical nature of the BAS) or on the ability to "integrate" different technologies into the management application suite (which we will discuss in the anti-pattern Section 2.2).

When considering the information technology (IT) point of view, which is not detailed in the handbook, we find that most proposed solutions are quite vertically-oriented, *i.e.*, they tightly integrate the application with the field devices. In this paper, we aim at analyzing the current IT design architectures and identify the major shortcomings and opportunities for improvement and growth. We will illustrate modern IT architectures that will allow a deeper integration of existing systems, overcoming the traditionally hard interoperability issues and enabling new use-cases to be more easily deployed in different kinds of buildings. These modern architectures are mostly based on independent middleware solutions and on explicit modeling of building and plant features: although different researchers propose different technical solutions, the general approach is shared. At the same time, however, we are witnessing an adoption delay of big players, who keep proposing vertically-oriented systems.

To clarify, the concept of "IT architecture," which should not be confused with the building's architecture from a construction point of view, represents the set of hardware and software components and systems that compose all of the computational nodes (servers, computers, user devices), all of the sensors and all of the communication infrastructure (wired or wireless, based on the Internet or on field-buses). An integral and important part of the IT architecture is the set of communications channels

between the mentioned components, the type of exchanged data and information and their relationships (master-slave, client-server or peer-to-peer).

The paper will start with the analysis of current (legacy) IT architectures commonly found in smart buildings, by defining a simplified high level that will enable us to identify and discuss limitations to their scalability. By adopting state-of-the-art architectural patterns developed in the fields of intelligent distributed systems and the nascent Internet of Things, the paper will present a modern, modular, interoperable, open and semantically-enabled solution. The key characteristics of the proposed architectures are: (1) the layering in horizontal levels, which may be combined according to system and application requirements and which tend to be independent from any specific application; and (2) the model-driven approach, where known information about the building, its devices, their functionality and interconnections are explicitly represented in a neutral, general, extensible and relationally-rich description by using semantic representation techniques, such as ontologies.

While many researchers and companies are currently working in such an updated IT scenario, there is still no agreement or alignment on the representation formats and standards; therefore, to illustrate the applicability of horizontal approaches, the paper describes some real examples that have been deployed by adopting the specific technologies (in particular we will adopt the DogOnt [2] ontology and the Dog Gateway) developed by the authors. The attained conclusions would be similar, if we analyzed some related approach.

The paper is structured as follows: Section 1 describes the proposed IT architecture, starting with the analysis of legacy architectural solutions in Section 2.1 and then outlining the structure and the benefits of horizontal layering in Section 2.3. A key aspect of horizontal architectures is their model-driven nature, which in our case refers to an ontology to describe the building environment and the controllable devices, represented as a working example by the DogOnt ontology shortly described in Section 4. The central part of the architecture is represented by the intelligent gateway node, and in our case, we describe the Dog Gateway in Section 5 as a concrete example of an implementation of the proposed architecture. To show the feasibility and to evaluate the advantages of open architectures, we also extensively present a case study, in Section 6, relating to a real building monitoring system, recently built using the living lab methodology. Finally, Section 7 concludes the paper and gives future directions.

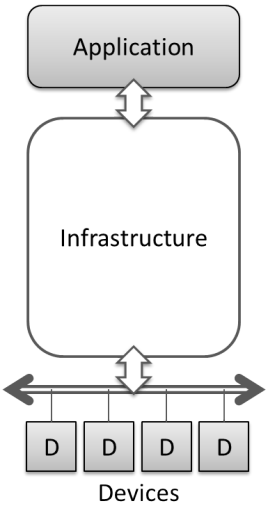
2. Horizontal Architectures

In this section, we examine the IT architecture of legacy building automation systems, from a high-level point of view, and we analyze their shortcomings in trying to scale to modern application requirements. Later, we present a new architectural approach that will overcome most of the current issues.

2.1. Common Approaches

By taking a sufficiently high-level view, the IT architecture of a BMS may be represented with the components shown in Figure 1: the system is composed of three layers (devices, infrastructure and application).

Figure 1. High-level view of legacy information technology (IT) architectures.



The application layer is responsible for fulfilling user-defined goals (where the user may be the building owner, energy manager or inhabitants): monitoring energy and environmental variables, remotely controlling some devices or subsystems, analyzing trends of historical data, applying sets of pre-defined computational models over real-time data, *etc.* Application-level technologies are usually deployed on computers or servers and may reside on-site or in the cloud. The application level should always include some sort of user interface (info panel, web portal, mobile application, *etc.*). Some examples of the issues tackled at the application level are:

- Creation of dashboards representing either the real-time situation or some overall performance indicator computed from current and past data observations. Such dashboards may be public or reserved to building managers. These are adopted mostly as a monitoring tool.
- Detection and handling of alarms (anomalous combinations of values, trespassing of thresholds, *etc.*) by constantly monitoring the systems and relaying the time- and security- or safety-critical information to the responsible person(s).
- Ability of remotely controlling some aspects of the system (such as operating actuators, activating or de-activating some automatic behaviors, adjusting some operational set-points, *etc.*) by means of a suitable user interface.
- Collection, storage and query of historical data, used as the consolidated basis of a comprehensive data set allowing more complex elaborations and analysis.
- Deep off-line analysis of the stored data, in order to extract trends, compare the performance in different periods and, more generally, define and compute sets of KPIs (key performance indicators) of interest to the system stakeholders. In some limited cases, computation may also involve real-time data (e.g., for threshold checking against historical averages), but it mostly consists of business intelligence methods applied to historical data sequences.
- On-line real-time computation of derived quantities [3]: in some cases, storing data and later elaborating it might waste resources (e.g., bandwidth from the building site to the cloud data center) and introduce unacceptable latency (e.g., for alarm management). In such cases, the on-site system will be required to compute some result, in real time, operating on fresh data: such computations might be simple (e.g., time averages, filtering outlier measurements, summing

up the contributions of different sensors) or more complex (involving computation of derivatives, integrals, the application of dynamic models or the emulation of virtual sensors).
- In some advanced cases, the complexity of the previous off-line and on-line computations, especially when integrated with user behavior or user interaction [4], may actually fall into the realm of ambient intelligence (AmI) [5,6].
- Finally, we should realize that a BMS could fruitfully exchange data with other information systems at the enterprise level, for example with ERP (Enterprise resource planning) systems for managing energy costs and billing [7] or with logistics software to get the expected occupation of building spaces. Such integration is usually a strong driver, which might justify the return on the investment of a BMS in the first place.

The first three items tend to cover similar functions to the SCADA (Supervisory Control and Data Acquisition) [8] systems found in industrial automation, while the latter are more akin to the software information systems domain.

The second level consists of the infrastructure layer, which aims at gluing together all involved devices, including sensors, actuators, computational nodes and the user interface, into a unified communication system. Such infrastructure is traditionally based on a wired infrastructure exploiting field-bus protocols, but increasingly, Ethernet-based or wireless solutions are being adopted. The main issues related to the design of the integration and communication infrastructure may be outlined as follows:

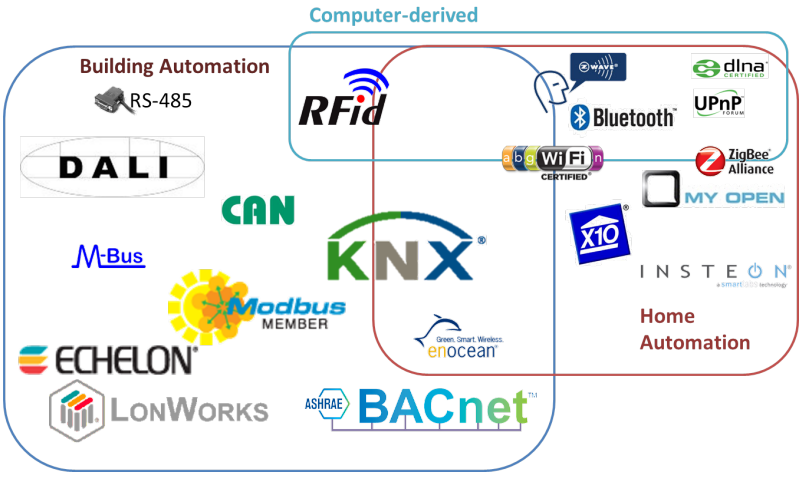
- The main driver behind the design of the communication infrastructure is, quite naturally, the technology chosen for field devices; see the relevant description below.
- The infrastructure is mainly identified by the communication protocols that are adopted; in fact, in many cases, protocols, devices and infrastructure are so intertwined that their choice must be joint. Such protocols may be derived from field-bus standards, from Internet standards or from wireless network solutions; in the worst cases, there may be legacy or proprietary protocols to be integrated in the infrastructures.
- Different design choices are also influenced by the scale of the installation: in fact, while the focus of the paper is on buildings, the same technologies and architectures may scale from very local solutions (e.g., houses), to wider and/or geographic-scale settings (buildings, sets of buildings, up to neighborhoods or smart city or transport systems).
- Related to the scale issue, also the number of devices may impact the choice of the infrastructure, both due to the maximum number of supported devices by each technology and due to the involved bandwidth necessary to transfer all needed data to and from all installed devices.
- The needed bandwidth is also influenced by the required sampling frequency: such frequency usually stems from application-level requirements (e.g., energy management systems usually work on a 15-min time scale, while automation and control should react in sub-second time) and is limited by the transmission capability of the chosen infrastructure (e.g., the transfer rate of long serial lines or the power saving needs of wireless sensors).
- An additional responsibility of the infrastructure layer is the enforcement of system security, by granting suitable authorizations to properly authenticated users and applications, only. When

dealing with field-buses, which are a classically closed and independent system, this issue was marginal, but when the systems are being integrated and being exposed to the Internet, such concerns become very demanding: the press constantly reports new home automation devices that can be hacked very easily.

- Depending on the type of devices and on the application requirements, the type of data being transferred may vary significantly: from simple Boolean variables or flags (e.g., in control systems), to real-valued physical measurements (e.g., temperature, humidity), to multimedia content (e.g., ambient sounds, environment images).
- The previous issue must be coupled with the problem of data encoding, which should guarantee that data is interpreted consistently by all involved devices and applications.
- Concerning the communication patterns, some simpler infrastructure technologies support a master-slave polling method, only, where the application must constantly and periodically query all sensors and devices. Conversely, more advanced infrastructures support the concept of time-driven or event-driven transmission, where devices may be pushing information to the application autonomously, as soon as new or updated information is available.

We may recognize that many of the infrastructure-level design choices are heavily dependent on the application requirements (which data is needed, how frequently, *etc.*) and on the chosen field devices. In fact, the many types of device categories and the diverse requirements for their adoption has given rise to a large number of infrastructure standards adopted in smart buildings: we collected the most popular ones in Figure 2, where we may appreciate the overlapping of different industry sectors fighting for attention in the smart buildings market: computer and consumer electronics, electrical plant providers and industry automation players.

Figure 2. Some commonly adopted building automation technologies.



In fact, the bottom level of the architecture consist of the devices layer, *i.e.*, a layer collecting all devices deployed in the environment. Such devices are mostly sensors or actuators, connected through the infrastructure level, but in some cases, the devices may also include user-interface elements (e.g., smart displays). The market for building automation devices is immense, and any user requirements may be usually satisfied by existing components, as new and more powerful ones are constantly being

produced. Of course, not all device types may be available in all integration technologies, and this forces design compromises and sub-optimal application-level performance, in some cases. To summarize, the available devices may be classified as:

- environment sensors, for sensing and measuring some physical property of a portion of the environment (temperature, humidity, CO_2, pollutants, illumination, wind, *etc*.);
- user sensors, for sensing and measuring some actions and conditions related to the usage of the building by users (presence, movement, access control, *etc*.);
- energy measurement devices (electrical energy, electrical power, more advanced electrical network properties, consumptions of gas, water, oil and other materials, *etc*.);
- actuators (relays, electro-valves, motors, lights, dimmers, *etc*.).

This list represents the "classical" set of devices, but more recently, the market has started to offer new types of connected components, which need somehow to be integrated in the building management system, even if they do not quite fit the existing architectures provided by mainstream integration standards. These new systems include, for example, smart appliances (washing machines, fridges, TV sets, *etc*.) that feature some sort of network connection (WiFi, BlueTooth or ZigBee, usually) and implement some custom functionality. A second example is IP-enabled surveillance cameras, which directly connect to the local area network (wired or wireless) and provide web-based image recording capabilities; such systems were born for small-scale home systems, but their cost effectiveness is starting to push them also to larger-scale buildings.

2.2. Common Approaches Pitfalls

From the above discussion, it should be evident that most critical issues arise in the definition of the infrastructure level, which bears most of the responsibility for supporting application requirements and, at the same time, managing a wide and possibly heterogeneous set of devices. By comparison, the applications tend to be either standardized (dashboards, data storage) or completely custom-built (such as intelligent control systems). However, the infrastructure level should accommodate the differences among different buildings and their adopted technologies. On the other hand, devices are seen as external components to be purchased, installed and configured, and no custom design usually is involved at this level.

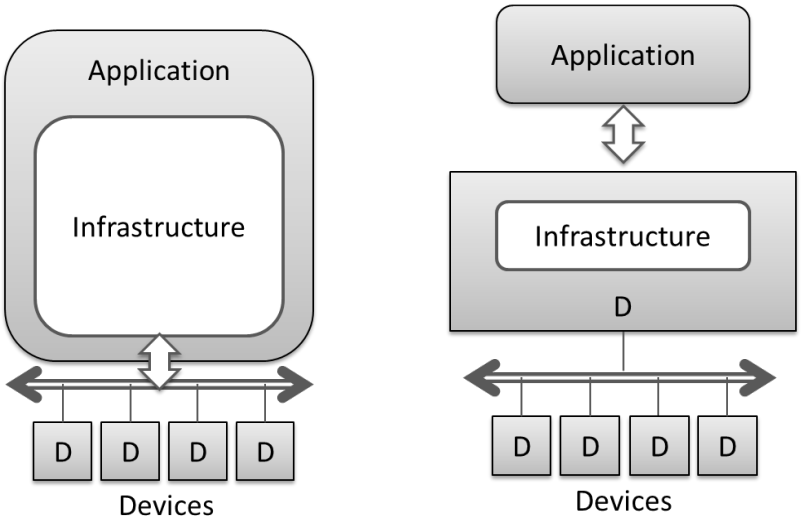
With the increasing complexity of building functions and the increasing variety of devices, the infrastructure level, which initially consisted of little more than the software to manage a network protocol, started inflating and incorporating new functionality. Among these, we may recognize: multi-protocol management, local caching or historical data when network connectivity is missing, support to mobile and/or disconnected devices, authentication and authorization of users, management of Internet connectivity and the associated routing, tunneling and proxying issues, seamless integration of master-slave and event-driven systems, the capability for on-line updates, just to mention a few of the dimensions along which complexity is increasing.

System designers and system integrators found and proposed different solutions to these issues, by finding a way to increase the power of the infrastructure level. However, their industrial background and their business models pushed them to architectural solutions that are suboptimal. The reader should not

infer that all proposed solutions fall in these categories, nor that all proposed solutions are not optimal in this respect; the paper tries to make the case for a new architectural approach and, therefore, analyzes two extreme, yet popular, approaches.

From a software engineering point of view, we may in fact identify two types of "anti-patterns", succinctly represented in Figure 3. An anti-pattern (or antipattern) is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive [9,10].

Figure 3. Scaling anti-patterns of IT architectures. (**a**) application handles infrastructure; (**b**) gateway handles infrastructure.



In this case, the detected anti-patterns are:

- "All you can eat" application (shown in Figure 3a). In this case, the infrastructure level has been "eaten" by the application layer. In other words, the additional complexity described above has been dealt with directly by the final applications. This approach, usually taken by system integrators or software-oriented companies, effectively solves the problems with a relatively moderate cost, since it relies on new software modules, libraries or functionalities. However, in the medium-to-long term, it makes applications impossible to port to different buildings: unless a new building is using exactly the same technology, the application will need to be re-customized by adapting its lower-level functions. Furthermore, if more applications need to be deployed in the same environment (e.g., the security system and the energy management system are both interested in knowing which rooms are occupied), then they need either to be part of the same application (creating single-vendor monopolistic situations) or the new application must replicate the infrastructure-level functionality of the first one (thus increasing overall costs and risking incompatibilities).
- The "tooooo smart" gateway (shown in Figure 3b). This approach relies on an additional (or more powerful) component, still at the device level (D), which incorporates the required complexity and is the preferred solution for hardware component manufacturers. These devices are usually called "smart gateways," "web servers," "controllers" or similar: they take the role of intermediating with

the underlying devices and offer high-level function to the applications. For example, these smart gateways may allow data collection, automatic sending of data via FTP (File Transfer Protocol) or e-mail, may embed a small web interface, may autonomously apply some simple logic (such as the one used by PLC—Programmable Logic Controller—devices) for sending commands, *etc.* Applications are simpler, since they can rely on the higher level functions offered by the gateway. Again, there are several disadvantages with this approach: first, it is limited to the set of devices that the gateway is able to handle, as new devices or other protocols are not supported at this level. Second, the offered functionality is not extensible (unless we hope for the device manufacturer to provide a new version with the needed extensions), as it is bound to the gateway implementation, and this can require some workarounds at the application level. Finally, the applications will be bound to the application programming interface (API) offered by the gateway: while the field-level communications among devices are standardized, these APIs are highly specific to each device and each manufacturer, and the development of the application will heavily rely on the specific API, thus creating a long-term commitment to reuse the application only in conjunction with the same gateway.

2.3. Semantics-Rich Architectures

The solution proposed in this paper and that is common to many of the related works (Section 3) for taming the complexity described in Section 2.2 and to the related anti-patterns is mainly based on recognizing that the additional functions do not belong to the application nor to the device level, and we must recognize them as being part of the infrastructure level. Instead of subsuming the infrastructure level onto the top or bottom layer, we must work to structure it better.

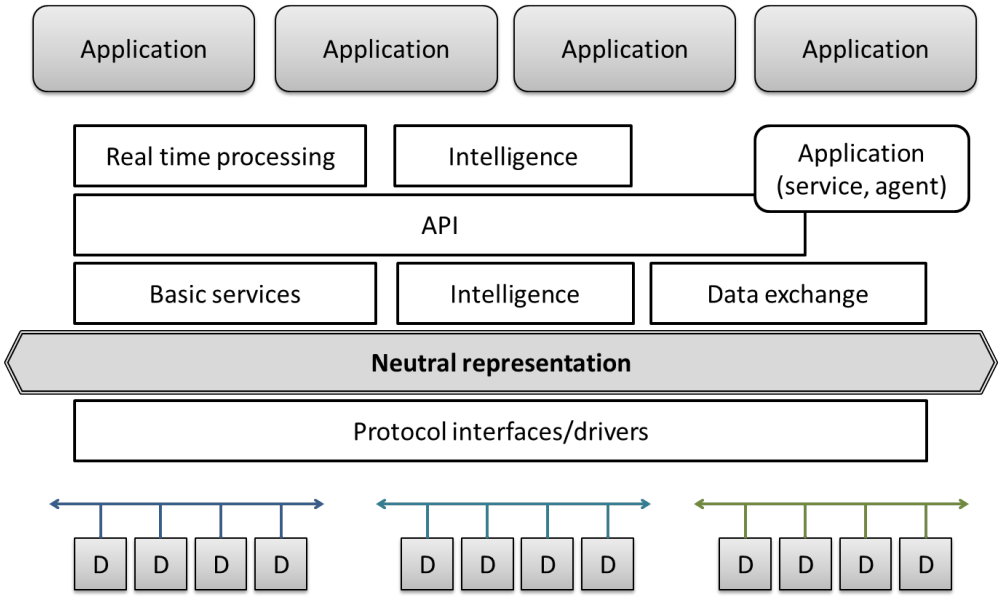
First of all, from a system integration point of view and in the interest of long-term development and future system evolutions, we should first solve the problems created by the anti-patterns: an excessive inter-dependency between application and devices. These legacy approaches may be termed "vertical" solutions, since they bind the whole system together. Smart buildings are created by putting many vertical systems side by side, with many applications, each using different sets of devices and communication infrastructures.

We propose to adopt an architecture based on "horizontal" layering, instead, as it is common in large enterprise software systems. A horizontal architecture should be able to decouple applications from devices (no direct interaction should be allowed), thus making applications as independent as possible from the technologies deployed in the field. On the other hand, the deployment of field devices should be as independent as possible from the specific applications (by still providing the necessary information and functionality according to user requirements, of course) and concentrate on the best possible integration, cost-effectiveness and quality of measurement and control. With these clear separations, all issues caused by the anti-patterns vanish, since applications may be added, updated or migrated anytime, and the field components can also be upgraded with little or no impact on the higher levels.

The key ingredient for enabling this "double-blind" architecture to effectively work is to rely on a model-driven approach, by representing in an explicit way all of the information that is needed by the software to run and to adapt to a particular configuration. In other words, the building functionality

is described in a "neutral" representation (see Figure 4) that effectively separates the various layers. This neutral representation contains information about what type of sensors, actuators and other devices are available, but splits this information into two sides: the bottom side (visible from the device level) includes all technology-specific information (such as protocols, device addresses, configuration information, *etc.*); and the top side (visible from the application level) only contains the functional information (what each device does, regardless of its technology). Application programmers will use APIs (Application Programming Interfaces) that relate to the abstract functional description, which will then be mapped to actual protocols and device commands.

Figure 4. Open and horizontal system architecture. API: application programming interface.



The proposed model-driven approach draws from concepts in software engineering, where a model-driven architecture is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as models. Model-driven software development [11] is currently a highly regarded development paradigm among developers and researchers. It is based on using domain-specific languages to create models that express application structure or behavior in an efficient and domain-specific way. These models are subsequently transformed into executable code by a sequence of model transformations.

In the specific case of smart buildings, in our previous work [12], we identified the domains of interest that ought to be modeled in order to support intelligent applications. Namely, the identified domains were: user, environment, actionable, controllable, AmI. In particular, the infrastructure layer that we are trying to model includes environment (the list of rooms and location, their connections), actionable (environment objects that are, or that can potentially be, actioned by means of one or more electrically controlled actuators, e.g., doors, windows and gates) and controllable (which includes all electrical devices that are either part of the building automation hardware, such as domotic plants or field buses, or that are integrated through a variety of protocols and communication means).

Describing the relevant information in these domains of interest requires the choice of a modeling language and formalism. In our case, we propose the adoption of semantic representation techniques, by

using the standards developed in the Semantic Web initiative [13], in particular ontologies expressed using the Ontology Web Language (OWL) [14]. In particular, we propose to adopt the DogOnt ontology [2], which covers the needed representation requirements and whose structure fits well with the patterns of object-oriented programming in distributed systems. A brief overview of the structure and content of DogOnt is given in Section 4.

Starting from the choice of an ontology (such as DogOnt) as the "neutral representation" depicted in Figure 4, we can define additional horizontal layers to enable the support of the needed functionality. The set of all horizontal layers constitutes a middleware software system that is able to effectively manage the infrastructure-level complexity. Such middleware is shared by all applications and is able to oversee all installed devices.

The main needed layers in the middleware are:

- Protocol interfaces/drivers. This is the only layer lying below the ontology representation. This level is the only one with actual knowledge of the hardware protocols and components available in the building and is able to translate between high level commands (such as kitchenLamp.On()) into low-level protocol-dependent messages (such as $*1*1*12\#\#$, an actual command for switching on Lamp Number 12 in the Open WebNet protocol). This layer may be composed of many drivers, one for each supported technology, thus offering the possibility of managing systems composed of various protocols (as suggested by the different groups of devices in the picture) in a seamless way.
- Basic services. On top of the neutral representation, some basic management services are needed, such as the ability to query the list of represented devices, their capabilities, their current (or last-seen) state and the discovery and configuration of new devices. Such services basically compose an object-oriented high-level wrapper around the semantically modeled devices, and hide most of the complexity of the ontology modeling.
- API. The services (basic and advanced) offered by the middleware should be easily accessible by applications, through a well-defined and possible stable application programming interface. Such an API defines all of the application-visible functionalities offered by the middleware, which in turn represent and extend the functionalities offered by the devices in the building. The same set of functions may be offered in different programming languages, by implementing proper API bindings (e.g., a Java binding, a C# binding, an HTTP (Hypertext Transfer Protocol) binding, a WebSockets binding, *etc.*), so that applications may be developed in different programming languages and may also be distributed in different computational nodes, but still have the same control capabilities.
- Real-time processing. In case some real-time processing is needed, this optional layer may provide stream processing capabilities (such as the ones described in [3]) to compute on-the-fly quantities, such as averages, counts, sums, thresholds and integrals, and, therefore, enable locally-generated messages (such as alerts) or reduce the amount of traffic sent to external applications (by transmitting aggregate data, only).
- Data exchange. This layer may optionally be adopted for storing, importing, exporting or querying data coming from the building sensors and actuators. Depending on the requirements, this could

imply storing locally a history of recently read values and/or sending to a remote server such data for further application-level processing.

- Intelligence. The availability of semantically-rich information about the building, its devices, their functionality and their current state may enable the implementation of "intelligent" behaviors (either by directly working on the ontology, which supports reasoning operations in OWL) or by using the defined APIs.

The stack of middleware layers may be used by applications, which may reside "outside" the middleware (*i.e.*, on a different server in the same Local Area Network (LAN) or across the Internet in a cloud computing center) or may be hosted "inside" (*i.e.*, working as an additional service or agent inside the middleware platform). Applications should be agnostic about specific devices and technologies and should only use the provided high-level APIs for getting information about the environment and for sending commands and receiving data. In this case, the same application may readily be applied to a different building, just by adapting the ontology to describe the new building's configuration.

As mentioned before, these abstract architectural layers must be implemented in some sort of middleware software. In our case study, we propose the adoption of the Dog Gateway [15] middleware, which will be described in some detail in Section 5. The design and implementation of the Dog Gateway closely follows the model-driven approach starting from domain ontologies. The reader interested in the software engineering processes involved in model-driven development of the Dog Gateway may refer to the recent paper [16].

3. Related Works

The approach based on explicit modeling and horizontal layering for smart building systems stems from the application of modern software engineering techniques, such as those developed for other enterprise information systems (especially web-based ones), where system complexity and the desire for flexibility and maintainability require a strong investment in the correct architectural choices [17,18].

At the same time, the variability in structure and in functional requirements for smart buildings make it difficult to efficiently reuse any given tool in a different building, unless most of such variability is handled externally to the software code. In this case, model-driven approaches come to the rescue, and in particular, semantic-web solutions, based on ontologies, are increasingly being explored.

For example, the European Union (Directorate General Connect) is fostering, in collaboration with ETSI (European Telecommunications Standards Institute) on the development of an initiative for consolidating an ontology for the uniform and standard representation (preliminary results and on-going efforts can be seen at [19] of smart appliances [20]. Similar approaches are ongoing for emerging machine-to-machine (M2M) technologies and are in the initial stage also for the smart city domain. all of these application domains have a strong overlap, as well as significant peculiarities and are all being addressed by adopting or developing ontological models.

We should notice that current building management systems, as described in [1], still do not apply these methods and are mainly based on custom and vertical approaches. However, all of the current functionality, at the user level, of current web-based BMS systems may be obtained by working at the application level in a horizontally-layered architecture.

The research directions in advanced architectures are mainly focused on two parallel streams: designing effective middleware solutions and creating and exploiting comprehensive semantic representations. These two streams are briefly presented below.

3.1. Middleware

To effectively operate, a smart building requires a software component named the middleware (or gateway) that represents the "intelligent" part of the system. Middleware is the computer software that connects software components or applications and provides services that enable communication and management of data in distributed applications.

In the smart building domain, it usually refers to the layer and software and services that separate communications with the actual sensors, actuators and technical plants from the software applications used for storing, analyzing data and providing user interfaces. This component is also typically responsible for merging the flow of data coming from the devices through their diverse networks and making them available to other software components present in the system, and it uses some Artificial Intelligence (AI) techniques for taking decisions or supporting the environment inhabitants. Moreover, an important task assigned to the middleware is to facilitate interoperability, *i.e.*, to help devices and networks created by different providers and speaking with different protocols to effectively understand each other and converge into a unique and uniform representation that can be easily understood by other software in the smart environment [21].

A solution similar to the one discussed in this paper is adopted in the UniDA (Uniform Device Access) framework developed at Universidade da Coruña (Spain) [22], which consists of a layered software approach, including a modular gateway powered by ontology information and running on a dedicated hardware platform. Incidentally, the adopted ontology is DogOnt, as in our current paper. The focus of the UniDA approach is more on the interaction with human users in the context of smart spaces.

Smart environment must be capable of changing their behavior dynamically based on user activities and the environment. To provide this, context-aware task-oriented middleware (CDTOM) was designed [23]. It provides service by abstracting the task from the routine activity with the intended goals. The CDTOM middleware provides various services, like context storage, context data acquisition, context-riven task reasoning, service discovery and task-oriented mapping.

The CASAS (Center for Advanced Studies in Adaptive Systems) [24] is a middleware platform that provides a lightweight design for a smart home that is easy to install and to provide the services of smart home capabilities without any customizations or training. CASAS is also referred as "Smart Home in a Box", with a lightweight infrastructure that can be extended with minimal effort. This kind of platform can be useful for building a targeted smart home in a specific context and disseminating smart home sensor data [25].

For accommodating legacy components, like transmitters and receivers, in the handover system, the middleware called PACE (Pervasive, Autonomic, Context-aware Environments) was developed [26]. In PACE, context and preferred repositories can be discovered dynamically with the help of mobile context-aware components using various service discovery protocols. The major problems tackled by the PACE middleware are security, scalability and fault tolerance.

The Hydra (Networked Embedded System middleware for Heterogeneous physical devices in a distributed arc hitecture) project [27,28] develops middleware for networked embedded systems that allows developers to create ambient intelligence applications based on wireless devices and sensors. Through its unique combination of service-oriented architecture (SoA) and a semantic-based model-driven architecture, HYDRA enables the development of generic services based on open standards.

Similarly, the recent work at Pondicherry University (India) aims at defining a generic middleware model for smart homes [29]: it is constructed with four layers, namely device, user, location and environment.

If we widen our focus from smart buildings to more general Internet of Things (IoT) systems, we find a similar architectural approach based on middleware systems. A current general survey about the architecture of IoT systems is given by Atzori *et al.* [30], while an overview of commonly adopted IoT middleware is described by Bandyopadhyay *et al.* [31].

The same approach is taken by Souza *et al.* [32], where they build smart home functionality by exploiting the Internet of Things, and by Perumal *et al.* [33], who propose an interoperability framework for smart home systems based on web services.

Another large effort in semantically-enabled middleware gateways is represented by the ThinkHome project [34,35] developed at Vienna University of Technology. The ThinkHome system operates on an extensive knowledge base that stores all information needed to fulfill the goals of energy efficiency and user comfort. Its intelligence is implemented as and within a multiagent system that also caters to the system's openness to the outside world.

Several funded research projects aim at consolidating and porting all of these research efforts into viable products and methodologies. Among them, let us cite the DEHEMS (The Digital Environment Home Energy Management System) project and its follow-up, SmartSpaces (Saving Energy in Europe's Public Buildings Using ICT) [36].

3.1.1. Programming Frameworks

all of the mentioned middleware solutions need to be based on solid programming frameworks, which allow the creation of modular, dynamic and distributed applications and services and which are open to many connectivity possibilities and data encoding formats. The actual adopted programming technologies vary a lot, from Java to .NET.

Among all possible programming frameworks on top of which gateways are constructed, the OSGi set of standards [37] surely stands out for its completeness and wide adoption. OSGi was originally meant to represent an "open service gateway initiative", but nowadays, they refer to themselves as "the dynamic module system for Java". The various implementations of the OSGi specifications (either open source or commercial ones) really help in managing all of the needed details in modern distributed applications and allow a level of flexibility and scalability that will never be reachable with the legacy architectures.

It is no surprise that many approaches found in the literature also adopt OSGi as the execution fabric. For more market-oriented information, the OSGi alliance publishes on its website the main products available in the smart home domain and also some interesting business success stories.

3.2. Semantic-Based Applications

While the focus of this paper is more devoted to the enabling nature of semantics-rich architectures as a core building block of intelligent building management systems, it is important to note that the same approach can be exploited in the various application domains. In other words, also the needs of end-user applications or domain-specific functionalities may be satisfied at the "application" layer (Figure 4) and still using a model-driven approach to extract technology-neutral information about the building from the underlying middleware.

Different examples of application domains are available; for instance, several works deal with the important issue of energy efficiency, either by automatic reasoning about the building status [38] or by developing suitable end-user interfaces [39].

Other approaches exploiting context-dependent semantic rules [40] or policy-based data management are being developed by Kumar *et al.* [41]. Policy-based discovery and triggering of semantic rules is also presented in [42], where the problem of conflicting policies is also explored.

Moving to larger contexts, such as smart cities, other application domains focus on the handling of the "big data" being generated by extensive collections of sensors [43], and semantics is again used to classify and filter the data.

4. The DogOnt ontology

All semantics-rich approaches must rely on an explicit domain model, which may be realized through an ontology or through other less flexible representations (e.g., taxonomies, standards). An overview of the currently active representations ontologies is given in [20]. One of the first ontologies in the smart home domain and one of the few that has been adopted by other research groups is DogOnt.

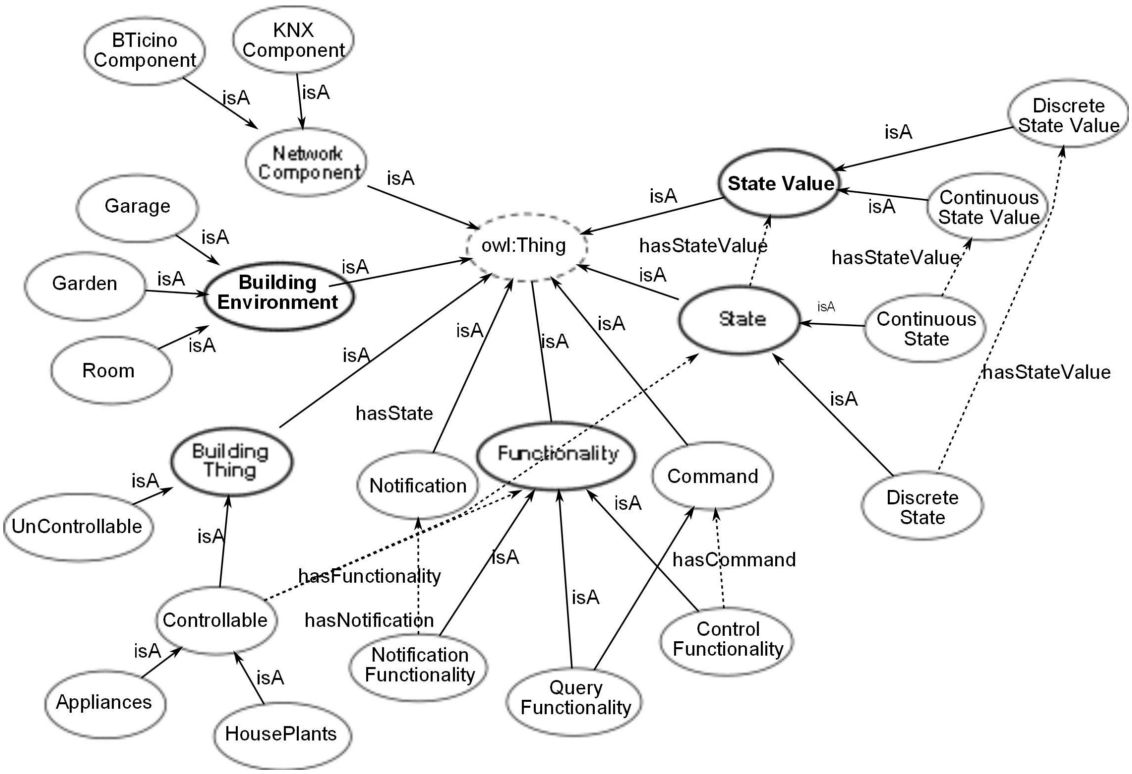
DogOnt is a domain ontology (*i.e.*, "an explicit specification of a conceptualization", as defined by Gruber [44]), specifically designed to model smart homes equipped with commercial domotic plants and intelligent appliances (for a complete description of the DogOnt design and modeling capabilities, see [2]) and whose representation can be stretched to model smart buildings, too. It is encoded in the OWL format (documentation about the DogOnt ontology is available at [45], while the OWL source can be downloaded at [46], as suggested by the W3C Semantic Web standard). The current version of the ontology is 3.2.10.

The design goals of DogOnt were to gather and integrate the best practices in device modeling, by merging the industrial standards with the emerging semantic representations. From the semantic point of view, the structure of DogOnt was strongly influenced by that of DomoML [47]. From the industrial world, the taxonomy of devices (controllables in DogOnt terminology) was designed to be compliant with the best practices developed by the European Home Systems Association (which is now part of the KNX specifications). More recently, also the home automation profile has been analyzed, and recent versions of DogOnt are aligned with ZigBee cluster definitions. Therefore, DogOnt may faithfully represent the device categories supported by industrial protocols and systems, but at the same time, tries to abstract common functionalities and allows querying and reasoning over the resulting ontology model.

DogOnt is currently exploited, among others, by the open source Dog Gateway [15], and it is organized along five main hierarchies of concepts (Figure 5, where hierarchy roots are represented in bold face), supporting the description of:

- The domotic environment structure (rooms, walls, doors, *etc.*), by means of concepts descending from BuildingEnvironment; they might be automatically generated starting from BIM (Building Information Modeling) data, if available, which is mainly useful for locating devices inside the building, by specifying the room or wall in which they are installed.
- The type of domotic devices and of smart appliances (concepts descending from the Controllable subclass of the BuildingThing main concept); this branch is the main one and is better described in the following.
- The working configurations that devices can assume, modeled by tates and StateValues; used to describe devices of the Controllable type; see the following paragraphs for more details).
- The device capabilities Functionalities) in terms of accepted events and generated messages, *i.e.*, Commands and Notifications; again, they are used to describe the behavior of Controllables.
- The technology-specific information needed for interfacing real-world devices NetworkComponent); the only branch where the encoded information depends on the details of the installed devices and should never be accessed by the upper layers of the architecture.
- The kind of furniture placed in the home (concepts descending from the UnControllable subclass of the BuildingThing main concept); this is a purely optional branch and might be useful for some kinds of spatial reasoning, which are outside the scope of the middleware.

Figure 5. DogOnt in a nutshell.



The core of DogOnt is the abstract modeling of devices (which are all located in a taxonomy rooted in the Controllable concept) in terms of functionalities and states.

Functionalities: These describe the device under the viewpoint of device interaction capabilities, *i.e.*, they describe how a given device can be controlled, queried and whether it can autonomously generate "events." For example, while a lamp can only be switched on and off, a light sensor can either be queried for the current luminance or can autonomously send luminance change events at regular time intervals. DogOnt functionalities include:

- ControlFunctionalities, modeling the ability of a device to be controlled by means of some message or command;
- QueryFunctionalities, modeling the ability of a device to be queried about its current state; and
- NotificationFunctionalities, modeling the ability of a device to issue notifications about state changes, in an event-driven interaction model.

Functionalities are either associated with commands (for ControlFunctionalities) or with notifications (NotificationFunctionalities) that further detail the specific operations supported by DogOnt device instances.

Device interconnections are modeled by the controlledObject relationship linking a controller device (e.g., a switch) to one or more controlled devices (e.g., a group of lamps). The same device can be involved in different connections with different roles, *i.e.*, as either a controller or a controlled device.

States: These describe the various stable configurations that a device can assume during its working life-cycle. From the modeling point of view, each device may include one or more different simultaneous behaviors. If we refer to a CD player, it can either be on or off, it can be playing a CD track with a given number and it may have a specific earphone output volume. In DogOnt, such behaviors are called States. The description of each State is represented by a set of identifiers, called StateValue, which model each operating condition. For example, the CD player is modeled as having three independent States: an OnOffState, a PlayingState and a VolumeLevelState. Each of these three states includes a specific set of possible state values (for example, the first state includes an OnStateValue and an OffStateValue). The current state of a device is therefore defined by a list containing one StateValue per each State.

5. The Dog Gateway

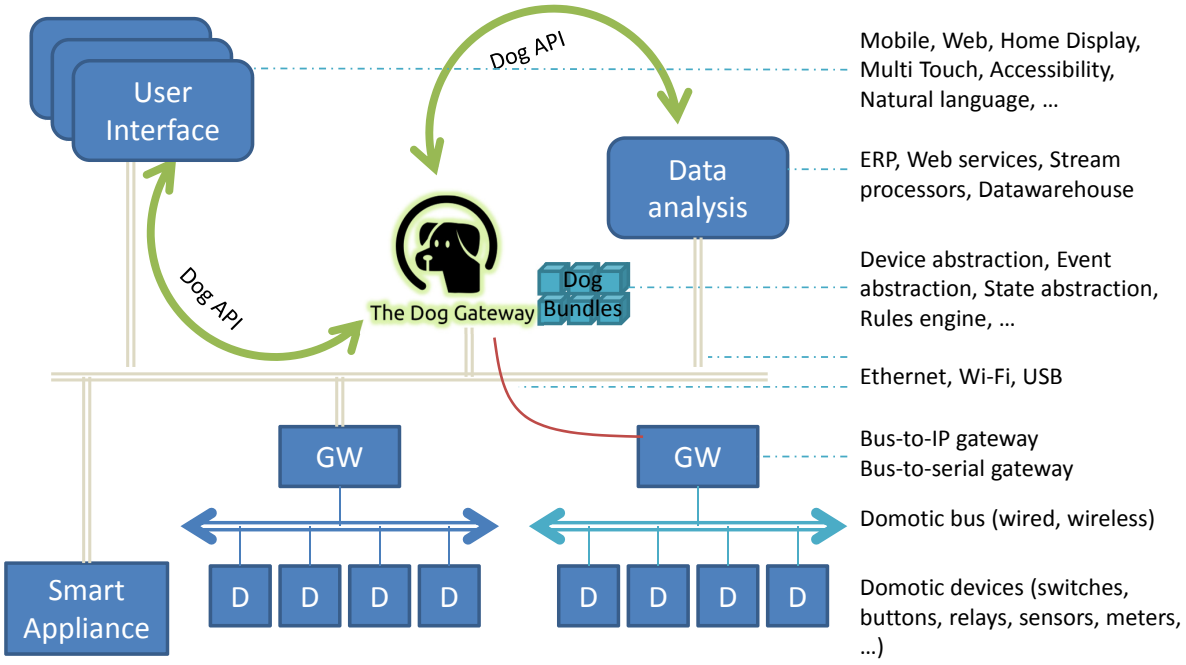
The core of an intelligent BMS is the central server, or gateway, or middleware, whose functions were already discussed in Section 3.1. The available gateways presented in the literature share some common traits, but each one possesses their specific strengths. In our work, the Dog Gateway has been selected for supporting the use cases and is briefly described in this section.

The Dog Gateway (formerly known as the Domotic OSGi Gateway [15]), as shown in Figure 6, is an ontology-powered middleware that exploits the OSGi framework [37] as a coordinator for supporting dynamic module activation, hot-plugging of new components and the reaction to module failures. Such basic features are integrated with the DogOnt ontology, for supporting the integration of different networks, to implement inter-network automation scenarios, to support logic-based intelligence and to access devices and appliances through an interface based on a neutral representation, according to our proposed horizontal semantics-rich architecture. Moreover, cost and flexibility concerns take a

significant part in the platform design: Dog is an open source solution capable of running on low cost (and low performance) hardware, such as a Raspberry Pi.

The Dog Gateway, born in 2008 with the name Domotic OSGi Gateway, is actively maintained and developed by the e-Lite research group at the Politecnico di Torino. During 2013, Dog has been improved in its design and underwent a code refactoring, which led to Dog 3.0, released with the Apache License 2.0 on GitHub [48].

Figure 6. Using the Dog Gateway as a middleware component.

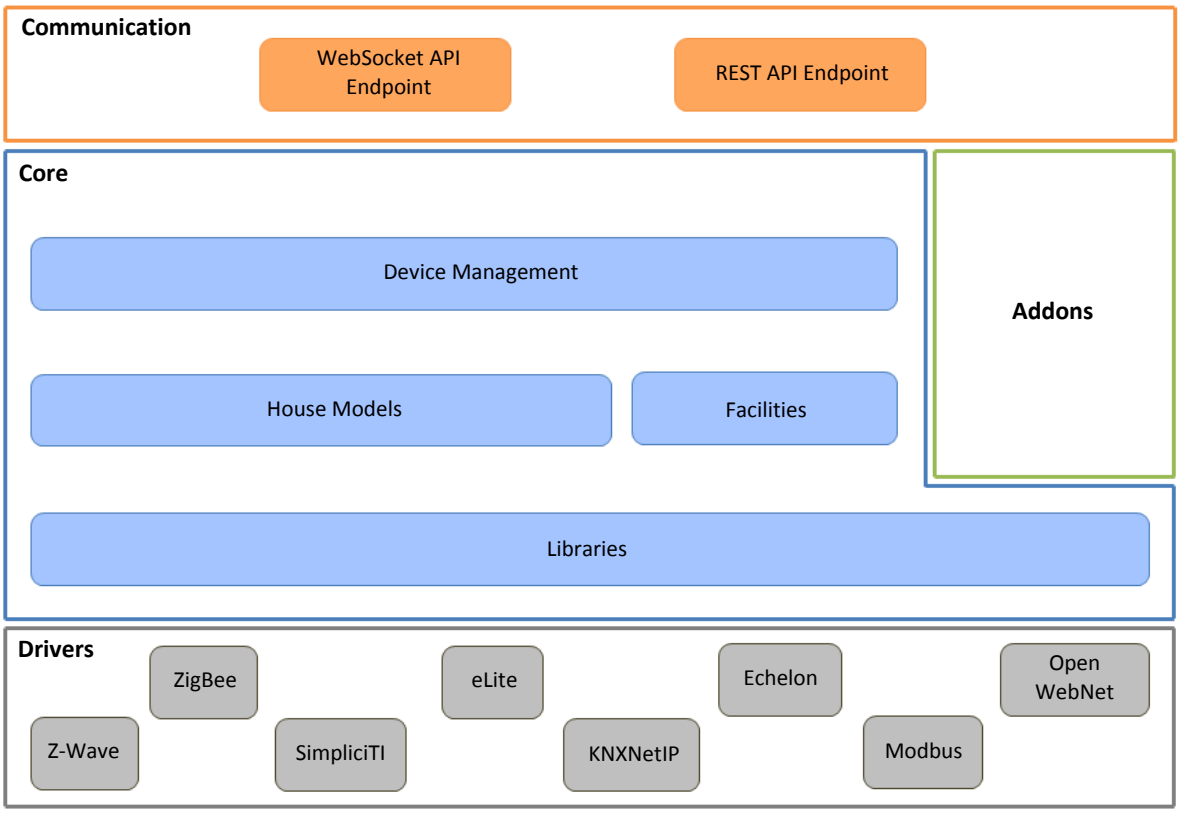


5.1. Architecture

The Dog design principles include versatility, addressed through the adoption of an OSGi-based architecture, advanced intelligence support, tackled by formally modeling the environment and by defining suitable reasoning mechanisms, and accessibility to external applications through a well-defined, standard API available in REST (over HTTP) and via WebSocket. OSGi (see Section 3.1.1) is a universal middleware that provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software life cycle, as well as remote management. It provides a general-purpose, secure and managed framework that supports the deployment of extensible services, known as bundles.

Dog is organized in a layered architecture with four layers, each dealing with different tasks and goals, ranging from low-level interconnection issues to high-level modeling and interfacing (Figure 7). Each layer includes several OSGi bundles, corresponding to the functional modules of the system.

Figure 7. Dog architecture.



The first layer, Drivers, encompasses the Dog bundles that provide an interface to the various home and building automation networks to which Dog can be connected. Each network technology is managed by a set of dedicated drivers, which abstract network-specific protocols into a common, high-level representation that allows one to uniformly drive different devices.

The second layer, Core, hosts the core intelligence of Dog, based on the DogOnt ontology, which is implemented in the semantic house model bundle. Moreover, it provides a set of common libraries and services useful to the entire systems or expected from the OSGi specifications.

The third layer, Addons, includes additional bundles for injecting further capabilities or more intelligence to the "core" part of the system, such as data storage, stream processing, rule engine, *etc.*

Finally, the fourth layer, Communication, provides the bundles offering access to external applications, either by means of a REST (Representational State Trasfer) endpoint or via WebSocket.

In the following, relevant services and functionalities of each layer are described in more detail.

5.1.1. Drivers' Layer

In order to interface home and building automation networks, Dog provides a set of drivers, one per each different technology (e.g., KNX NetIP, Z-Wave, *etc.*). Driver implementation and operation follow the OSGi Compendium Specification [37]. Each driver implements a "self-configuration" phase, in which it interacts with the OSGi framework to retrieve all of the needed low-level information, according to the specific technology, e.g., the device address(es) or its ID in the network. For each technology, a Network Driver, a Gateway Driver and at least one Device Driver must exist.

The Network Driver handles network-level communication, in terms of protocols, connections and polling (when needed). It defines also the network access APIs for all of the Device Driver bundles of the same technology. In Dog, only one Network Driver may exist for each technology.

The Gateway Driver supports multi-gateway operation for a given technology and handles the association between devices and their gateways. It permits one to install Device Driver bundles if and only if the corresponding network gateway is present at the configuration level. Moreover, according to the specific technology, it can provide gateway-specific commands and functionalities. In Dog, only one Gateway Driver may exist for each technology.

A Device Driver implements the DogOnt device features for a given class of devices, *i.e.*, it translates ontology-defined commands, functionalities and states into network-level messages. Typically, in Dog, a Device Driver is created for each DogOnt device class.

Currently, eight different technologies are supported by Dog, each with a different number of device categories covered: KNX NetIP, Modbus (RTU (remote terminal unit) and TCP (transmission control protocol)), Echelon iLon100, eLite (*i.e.*, a set of simulated drivers), BTicino OpenWebNet, Z-Wave, ZigBee Home Automation profile and SimpliciTI.

5.1.2. Core Layer

The Core Layer encompasses the core intelligence of Dog and provides a set of common libraries and services useful to the entire system or expected from the OSGi specifications.

The Device Management category comprises three bundles for handling all of the life-cycle of a Device and its status variables.

The Device Factory bundle is responsible for creating and destroying Device instances in the framework, according to the runtime configuration it receives. Such a configuration can be provided by one of the house models or, optionally, injected by a gateway driver, if the underlying network supports the runtime discovery of new devices.

The Device Manager bundle implements the OSGi Device Access Specification [37]: it manages the procedure for matching and attaching a Device to the "right" Driver (if any), each time one of them is added, modified or removed in the framework.

The Monitor Admin bundle implements the OSGi Monitor Admin Service Specification: it provides unified access to any declared status variables defined in the framework. Moreover, it offers security checking and scheduling of periodic or event-based monitoring jobs, *i.e.*, it sends events related to some specified status variables according to defined rules.

The House Models category encompass two bundles: the Semantic and the Simple House Model.

The Semantic House Model manages the building (or home) description in the form of DogOnt instances, thus supporting model merging, implementing classification and basic reasoning, supporting the extraction of interoperation rules and, potentially, providing access to all of the properties defined in the ontology. Moreover, it can generate the XML configuration to be used by the Simple House Model.

The Simple House Model manages the building description in XML, and it is typically used for Dog instances targeted at devices with low computational power. It does not have any model merging capabilities or reasoning support.

The Facilities category provides two bundles: a Logger and a Clock. Such bundles offer, respectively, a console logger through the OSGi LogService, and an internal clock service that triggers a time event each second.

The Libraries category consists of six bundles that act as repositories of classes, interfaces and various services needed by the other Dog bundles. The JAXB Library (Java Architecture for XML Binding) provides XML serialization and de-serialization for handling the Simple House Model configuration and for some messages provided by the Communication Layer. The Measure Library provides units of measure and related operations by offering the JScience library to all Dog bundles; it defines also some units of measure unsupported by JScience. The Semantic Library encapsulates and makes available all semantic-related libraries, such as Apache Jena, Pellet and a SPARQL (SPARQL Protocol and RDF Query Language) query facilitator. The Stream Library defines some types of events to offer unified access to them for uniform handling by the Complex Event Processor that can be used in an Addons bundle. The org.rxtx, similarly to the Measure Library, exports the serial port API library to all Dog bundles.

Eventually, the Core Library is the most important:

- It contains all of the possible devices, functionalities, states and state values as defined in DogOnt; all of these classes and interfaces are programmatically generated from DogOnt, thus ensuring a formal and full compliance with the ontology representations.
- It encompasses all of the possible device implementations, defined as abstract classes and programmatically generated from the previous part.
- It provides core-level notifications, *i.e.*, not defined in DogOnt, as well as common data structures needed by the other Dog bundles.
- It provides utility classes to other bundles, to avoid code duplications and repetitions.

5.1.3. Addons Layer

This layer provides additional bundles for injecting further capabilities or more intelligence to the previous layer. Currently, it comprises the following bundles.

The Rule Engine bundle provides a rule engine runtime for defining automation scenarios, interoperation and complex device behaviors; it is programmable through dedicated XML messages, coming from the Communication layer or read from a disk. It uses notifications as triggers, states as constraints and commands as rule-consequent actions.

The Power Bundle offers power consumption estimation based on actual measures, typical or nominal values defined in the DogOnt power extension and exploited by thePower Model. This model provides power-specific query functionalities and plugs into the Semantic House Model.

The Stream Processor bundle provides stream processing capabilities by handling measure and Boolean events for processing them in the spChains framework [3,49].

Finally, the Event Storage bundle maintains a storage of relevant events (e.g., measures) by using an in-memory database with a small footprint. Memorized data can be then visualized and analyzed according to any specific need.

5.1.4. Communication Layer

The Communication Layer provides access from external, non-OSGi applications by offering two alternative endpoints: a REST API and a WebSocket API.

Both APIs use the same basic message structures and expose the same functionalities and information: they help retrieving the building configuration, sending commands to devices managed by Dog, handling the building structural information (rooms, flats, *etc.*), getting the devices status, *etc.* The only exception concerns asynchronous events (*i.e.*, notifications), which may come from any device handled by Dog: they are provided only by the WebSocket endpoint due to the synchronous nature of HTTP.

5.2. Implementation

The Dog Gateway has been implemented in Java, as a set of 88 self-developed OSGi bundles running on the Equinox OSGi implementation. Additional, but mandatory, bundles, implementing standard OSGi services, are taken from diverse OSGi open source implementation and mainly from Equinox and Apache Felix.

The DogOnt ontology is managed by the Semantic House Model using the Apache Jena API, while the external API modules exploit the JAXB (Java Architecture for XML Binding) project for handling XML contents and the Jackson JSON Processor for managing JSON documents. The REST endpoint uses the Jersey RESTful Web Services framework, which provides a good toolkit for developing RESTful Web Services in Java.

The current version of the Dog Gateway (3.0) is released on GitHub, in both binary and source formats [48], under the Apache License 2.0, while previous versions of the gateway were released on SourceForge. Dog runs on very cheap computers, such as the Raspberry Pi: a credit-card sized computer with an ARM processor at 700 MHz and 512 MB of RAM.

6. Case Study

In Section 2.3, we described a methodology for designing smart buildings, exploiting horizontally-layered IT architectures, model-driven development approaches and semantics-rich representation. Such a methodology is the current focus of many research initiatives and led to the development of various ontologies and middleware products, outlined in Section 3.

To illustrate from a practical point of view the benefits of such approaches, we briefly present the result of a recent project, where such guidelines have been followed and where the DogOnt ontology and the Dog Gateway middleware have been adopted.

In 2013, the authors, in collaboration with a local SME, called Proxima Centauri, participated in a pre-commercial project published by Regione Autonoma Valle d'Aosta, a small region in the north-west part of the Italian Alps. The project, called "Applus.énergie" (in French: "apprendre plus, consommer moin énergie", *i.e.*, "learning more, consuming less energy"), aimed at creating innovative energy monitoring systems and validating them using a living labs methodology. For the project, we selected to work on two different educational institutions, hosted in peculiar buildings:

- One building in the town of Verrès (in the Aosta region), Italy, which hosts a secondary technical school and a university college degree program. The building, shown in Figure 8, was renovated around 10 years ago, starting from an old abandoned cotton factory; it features a large central gallery (closed with glass windows on the top) and seven stories of balconies, which host classrooms, laboratories and offices. The building has a state-of-the-art climate control system, featuring air control units for both classrooms and for the central gallery; the electrical system is well built, but featured no automation nor control.
- One building in the city of Torino, Italy, in the area called Mirafiori, which hosts a part of the campus of the Politecnico di Torino university. The building, shown in Figure 9, was totally renovated in 2011 staring from a group of production buildings formerly owned by FIAT Auto. The building contains classrooms, laboratories and some offices and features a climate control system and an electrical control and monitoring system, both quite modern, but completely separated (a nice example of side-by-side vertical architectures).

Figure 8. Monitored building in Verrès (Aosta), Italy.

Figure 9. Monitored building in Torino, Mirafiori, Italy.

Our design goal was to build an energy-monitoring system (including both electrical and thermal forms of energy), for both buildings, and to provide user interfaces that, according to the living labs methodology, could increase user awareness about energy consumption and possibly foster virtuous

behaviors. As the test sites, we selected technical schools, because we believe that such students are the most likely targets to understand the interactions between energy, behavior and technical plants and that, in their family and future professional life, they can spread positive changes.

As always, budget was tight and timing was even tighter. Additionally, if we had applied legacy design methodologies, we would have faced several obstacles:

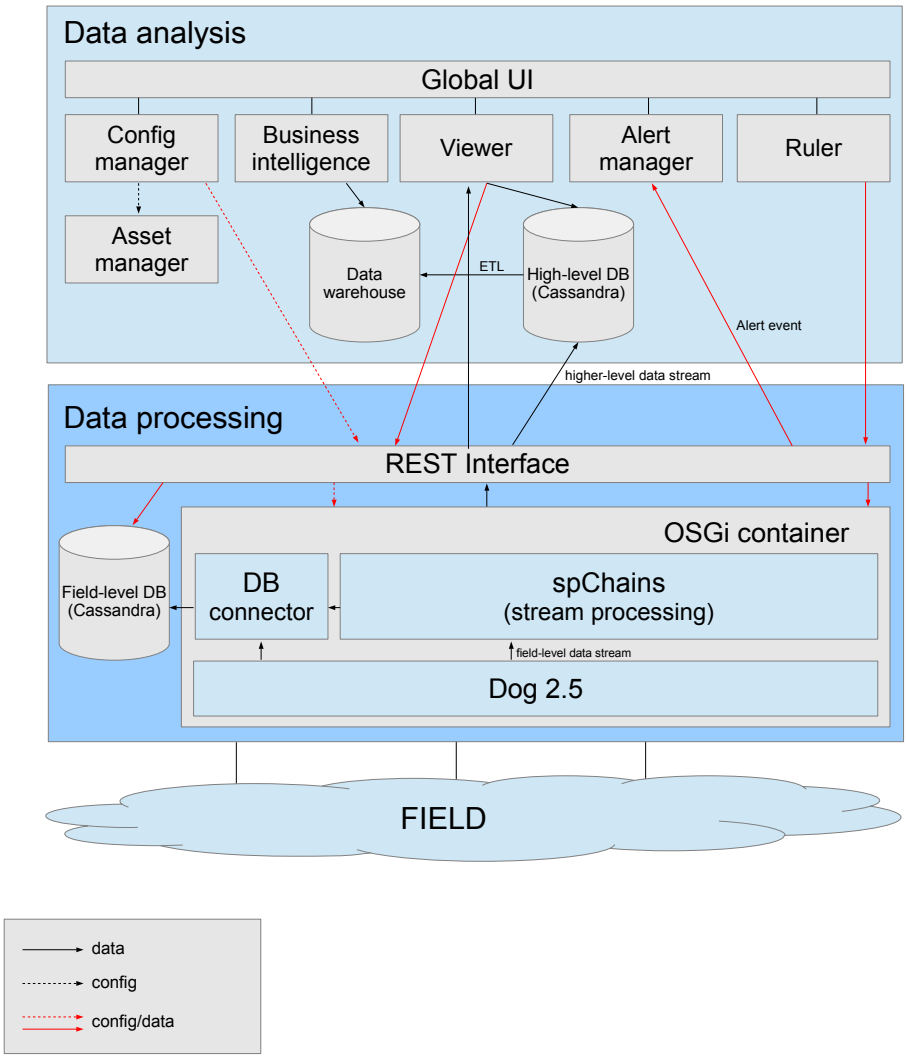
- In Verrès, there was no electrical monitoring system. This is sort of an easy case, since we had then the possibility of choosing the most suitable technology. It was not easy to pull serial lines across the electric plant, but there were free Ethernet lines running nearby: we decided to install new electricity meters and send their data over the local Ethernet LAN. We chose ModBus meters encapsulated in RTU over IP.
- In Verrès, the air and temperature control system was built using a proprietary system, with no documentation available for connecting. However, the system was already measuring a lot of temperatures and humidity values across the whole building (to feed its control algorithm), so we decided to try and re-use such information without adding new environmental sensors. With the collaboration of the company managing the plant, we asked them to install a converter from their protocol into ModBus, so that a subset of the sensor values could be read over ModBus/TCP.
- In Verrès, the huge central gallery did not have sufficient environmental sensors (the variables provided by the control system were too sparse), so we decided to add new sensors. Due to the size and the construction of the gallery, it would have been difficult to install new wired sensors, so we opted for wireless sensors adopting the ZigBee protocol.
- In Mirafiori, the electricity distribution was already metered, thanks to a ModBus system, and it already had TCP gateways: it was therefore easy to get the needed information using the ModBus/TCP protocol, with the collaboration of the system builders.
- In Mirafiori, also, the thermal control was automated, although using a completely different wiring and gateway. Due to difficulties of communication with the system builders, we resorted to extract (in a periodic and automatic way) the necessary information from the management software that was running on a control PC.
- In Mirafiori, we wanted to have some more detailed information about the behavior of the students in a study room (where they can plug in their PCs, control lights and blinds), so we added additional sensors (light, temperature, humidity and electrical consumption on a desk-by-desk basis). Again, we adopted a mixture of ZigBee and ModBus protocols.

As we see, we decided how to "instrument" the field, by taking into account the general application requirements (which data are needed) and by trying to select the best possible technology according to what was already installed and to logistic constraints. We were confident that the horizontal architecture would support application development independent of component selection; therefore, we had a much wider space for optimizing design choices.

In parallel to sensor planning and deployment, we defined the IT architecture, trying to follow the model-driven horizontally-layered semantically-rich approach. The deployed architecture is depicted in Figure 10 and conforms to the general structure of Figure 4. The figure shows the complete separation between the applications, the middleware and the field. The middleware infrastructure layer (labeled

"data processing") is heavily based on the Dog Gateway (in its former version, 2.5), with the addition of a real-time stream processing module (spChains [3]) and of a local database for storing historical data. The application level (labeled "data analysis") queries the middleware layer through a programming interface implemented using the REST conventions over the http protocol (the Dog APIs have been wrapped in REST/HTTP). Later, the application stores in a local database a copy of data extracted from the middleware and later analyzes it with data warehouse and business intelligence methods. Finally, the application provides some user-accessible dashboards.

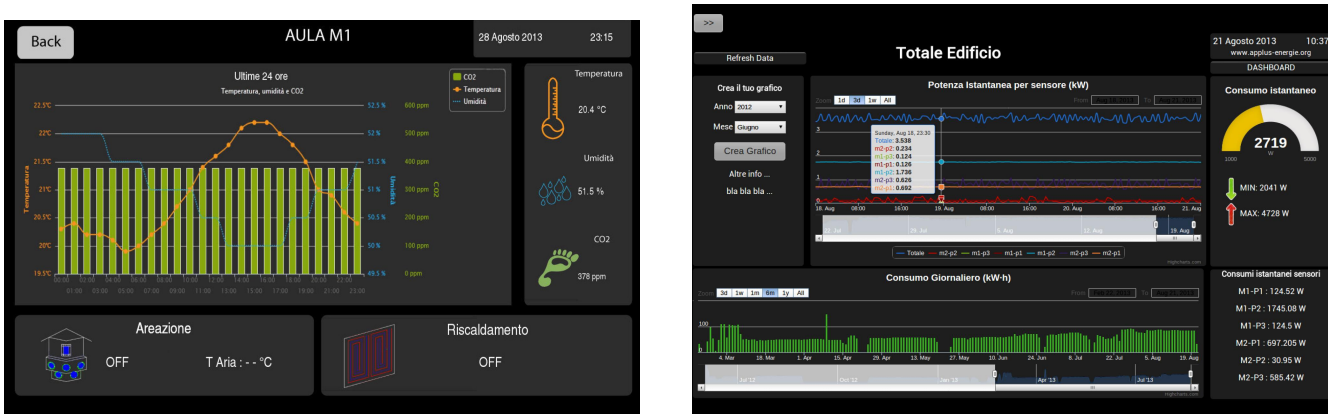
Figure 10. IT architecture for the Applus project.



We should stress that the same software (including middleware and application) has been deployed in the two buildings, even if their characteristics and their installed sensors were completely different. The only difference was in configuration information (the DogOnt description of buildings and the real-time filters to be applied by spChains). We also want to highlight that, differently from legacy approaches, all data analysis and user interfaces lie at the application level, so any change to the application level or to the user interfaces would have no impact over the lower level. Furthermore, the possible addition of new sensors at the field level would have a minimal impact on the middleware and application ones.

The user awareness about the system was provided thanks to a series of public dashboards (Figure 11), which were also shown on smart-screens positioned in strategic places inside the building (*i.e.*, the main entrance, the coffee machines, the PC labs, *etc.*). After building and deploying the architecture, we started the living lab activities, by involving students, organizing focus groups and specific class lectures about the system. A selected group of students also had access to the raw data acquired in real time, and they did some homework projects by analyzing it. The experience with the students was very positive, and we gathered significant feedback that helped us to improve the dashboards and to imagine the impact on energy consumption.

Figure 11. Example monitoring dashboards.



6.1. Lessons Learned

In the development of our pilot case studies, we aimed at following the guidelines that were set forward in Section 2.3, to gain some hindsight into the theoretical and practical advantages and disadvantages of the semantics-based approaches with respect to traditional systems.

The overall functionality of the systems were compatible with those offered, albeit with a significant price tag, by commercial vendors. We evaluated that we could reach a functionally complete prototype in a relatively short time, since most of the activity was limited to:

- Adding new "Driver" modules to compensate for new protocols encountered by pre-existing BMS systems (and this work would capitalize on future implementations).
- Modeling the set of devices and the building configurations, by creating suitable sets of semantic instances.
- Defining the streaming computation chains to compute significant indicators starting from raw measurements.

In other words, most of the work was declarative modeling, with a limited amount of custom programming. In fact, the two pilots used significantly different technologies and devices, but the running software was identical, and only the configuration data were different. This application of a strongly model-driven approach allowed us to tackle, in a matter of a couple of months, complex systems that involved at least four different protocols (Modbus, ZigBee, C-Bus and a custom 433-MHz wireless link), mixing pre-existing devices with newly installed ones.

Our experience shows that semantic approaches, after a significant up-front investment, may allow lean and efficient implementation of building intelligence solutions using open source software, thus reducing the overall investment costs and returning the control to the hands of the building owners. This conclusion is in line with the findings highlighted by the various research projects (see Section 3) that have adopted a semantic modeling approach.

7. Conclusions

This paper advocates the use of three main design methodologies—horizontally-layered IT architectures, model-driven development approaches and semantics-rich representation—and argues for the medium- and long-term benefits of such approaches, especially when independence from devices (or a variety of device technologies) or when application portability are important requirements.

Starting from previous research work on model-driven development, in particular on semantic modeling of smart buildings with DogOnt and from the experience in building and applying the Dog Gateway, we formalized the general architectural IT requirements that could boost the design of smart buildings and, in particular, allow an easy integration of subsystem and the enhancement of application functionality. Such general requirements are common to many other research trends, as outlined in Section 3, but are not yet adopted in the design of mainstream building management systems [1].

Finally, we demonstrated how the methodology has been applied in building the energy monitoring system of two very different buildings, by easily accommodating their peculiarities in the semantic modeling and, therefore, gaining effectiveness in the application development and system maintainability.

Acknowledgments

This work has been partially supported by the Italian company, Proxima Centauri, in the context of regional innovation projects funded by Regione Piemonte and by the Regione Autonoma Valle d'Aosta in the context of the Alcotra Innovation cross-border initiative. The authors wish to thank all collaborators who worked on the Dog Gateway project's community and all of the partners of the Applus.énergie project.

Author Contributions

Dario Bonino: ontology definition and gateway development; Fulvio Corno: paper writing and project coordination; and Luigi De Russis: living lab management and gateway development.

Conflicts of Interest

The authors declare no conflict of interest, other than being deeply and genuinely enthusiastic about the field of intelligent buildings and the potential for new applications and capabilities to empower users and managers.

References

1. Capehart, B.L.; Middelkoop, T. *Handbook of Web Based Energy Information and Control Systems*, 1st ed.; Fairmont Press: Lilburn, GA, USA, 2011.
2. Bonino, D.; Corno, F. DogOnt—Ontology Modeling for Intelligent Domotic Environments. In *Proceedings of the Semantic Web—ISWC 2008*, Karlsruhe, Germany, 26–30 October 2008 ; Sheth, A., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K., Eds.; Springer-Verlag: New York, NY, USA, 2008; pp. 790–803.
3. Bonino, D.; Corno, F.; de Russis, L. Real-Time Big Data Processing for Domain Experts, An Application to Smart Buildings. In *Big Data Computing/Rajendra Akerkar*; Taylor & Francis Press: London, UK, 2013; Volume 1, pp. 415–447. ref.2
4. De Russis, L. Interacting with Smart Environments: Users, Interfaces, and Devices. Ph.D. Thesis, Politecnico di Torino, Torino, Italy, 2014.
5. Cook, D.J.; Augusto, J.C.; Jakkula, V.R. Ambient intelligence: Technologies, applications, and opportunities. *Pervas. Mob. Comput.* **2009**, *5*, 277–298.
6. Sadri, F. Ambient intelligence: A survey. *ACM Comput. Surv.* **2011**, *43*, 1–66.
7. Bonino, D.; de Russis, L.; Corno, F.; Ferrero, G. JEERP: Energy aware enterprise resource planning. *IT Prof.* **2014**, *16*, 50–56.
8. Boyer, S. *SCADA: Supervisory Control and Data Acquisition*; International Society of Automation: Durham, NC, USA, 2010.
9. Neill, C.J.; Laplante, P.A. *Antipatterns: Identification, Refactoring, and Management*; CRC Press: Boca Raton, FL,USA, 2005.
10. Koenig, A. Patterns and Antipatterns. In *The Patterns Handbook: Techniques, Strategies, and Applications*; Cambridge University Press: Cambridge, UK, 1998; pp. 383–390.
11. Völter, M.; Stahl, T.; Bettin, J.; Haase, A.; Helsen, S. *Model-Driven Software Development: Technology, Engineering, Management*; John Wiley & Sons : Hoboken, NJ, USA, 2013.
12. Bonino, D.; Corno, F. DoMAIns: Domain-based modeling for ambient intelligence. *Pervas. Mob. Comput.* **2012**, *8*, 614–628.
13. Berners-Lee, T.; Hendler, J.; Lassila, O. The semantic web. *Sci. Am.* **2001**, *284*, 34–43.
14. Mcguinness, D.L.; van Harmelen, F. *OWL Web Ontology Language Overview*; W3C: Cambridge, MA, USA, 2004.
15. Bonino, D.; Castellina, E.; Corno, F. The DOG gateway: Enabling ontology-based intelligent domotic environments. *IEEE Trans. Consum. Electron.* **2008**, *54*, 1656–1664.
16. Bonino, D.; Procaccianti, G. Exploiting semantic technologies in smart environments and grids: Emerging roles and case studies. *Sci. Comput. Programm.* **2014**, *95*, 112–134.
17. Vernadat, F. Enterprise Modelling and Integration. In *Enterprise Inter- and Intra-Organizational Integration*; Kosanke, K., Jochem, R., Nell, J., Bas, A., Eds.; Springer: New York, NY, USA, 2003; Volume 108, pp. 25–33.
18. Niu, N.; Xu, L.D.; Bi, Z. Enterprise information systems architecture-analysis and evaluation. *IEEE Trans. Ind. Inform.* **2013**, *9*, 2147–2154.

19. Smart Appliances Study SMART 2013/0007. Available online: https://sites.google.com/site/smartappliancesproject/ (accessed on 31 October 2014).
20. Den Hartog, F.; Daniele, L.; Roes, J. *Study on Semantic Assets for Smart Appliances Interoperability—D-S1: FIRST INTERIM REPORT*; Technical Report for TNO Innovation for Life : Brussels, Belgium, 2014.
21. Nakashima, H.; Aghajan, H.; Augusto, J.C. *Handbook of Ambient Intelligence and Smart Environments*, 1st ed.; Springer: New York, NY, USA, 2009.
22. Varela, G.; Paz-Lopez, A.; Becerra, J.A.; Vazquez-Rodriguez, S.; Duro, R.J. UniDA: Uniform device access framework for human interaction environments. *Sensors* **2011**, *11*, 9361–9392.
23. Ni, H.; Abdulrazak, B.; Zhang, D.; Wu, S. CDTOM: A context-driven task-oriented middleware for pervasive homecare environment. *Int. J. UbiComp* **2011**, *2*, 34–53.
24. Cook, D.; Crandall, A.; Thomas, B.; Krishnan, N. CASAS: A smart home in a box. *Computer* **2013**, *46*, 62–69.
25. Cook, D.; Schmitter-Edgecombe, M.; Crandall, A.; Sanders, C.; Thomas, B. Collecting and Disseminating Smart Home Sensor Data in the CASAS Project. In Proceedings of the CHI09 Workshop on Developing Shared Home Behavior Datasets to Advance HCI and Ubiquitous Computing Research, Boston, MA, USA, 4–9 April 2009.
26. Henricksen, K.; Indulska, J.; McFadden, T.; Balasubramaniam, S. Middleware for Distributed Context-Aware Systems. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*; Springer: New York, NY, USA, 2005; pp. 846–863.
27. Eisenhauer, M.; Rosengren, P.; Antolin, P. A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence Systems. In Proceedings of the 6th Annual IEEE Communications Society Conference on SECON Workshops' 09, Rome, Italy, 22–26 June 2009; pp. 1–3.
28. Eisenhauer, M.; Rosengren, P.; Antolin, P. Hydra: A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence Systems. In *The Internet of Things*; Springer: New York, NY, USA, 2010; pp. 367–373.
29. Madhusudanan, J.; Hariharan, S.; Selvan, M.A.; Venkatesan, V.P. A generic middleware model for smart home. *Int. J. Comput. Netw. Inform. Secur.* **2014**, *6*, 19–25.
30. Atzori, L.; Iera, A.; Morabito, G. The Internet of things: A survey. *Comput. Netw.* **2010**, *54*, 2787–2805.
31. Bandyopadhyay, S.; Sengupta, M.; Maiti, S.; Dutta, S. A Survey of Middleware for Internet of Things. In *Recent Trends in Wireless and Mobile Networks*; Özcan, A., Zizka, J., Nagamalai, D., Eds.; Springer: Berlin-Heidelberg, Germany, 2011; Volume 162, pp. 288–296.
32. Souza, A.M.; Amazonas, J.R. A Novel Smart Home Application Using an Internet of Things Middleware. In Proceedings of the 2013 European Conference on Smart Objects, Systems and Technologies (SmartSysTech), Erlangen, Germany, 11–12 June 2013; pp. 1–7.
33. Perumal, T.; Ramli, A.; Leong, C.Y. Interoperability framework for smart home systems. *IEEE Trans. Consum. Electron.* **2011**, *57*, 1607–1611.

34. Reinisch, C.; Kofler, M.; Kastner, W. ThinkHome: A Smart Home as Digital Ecosystem. In Proceedings of the 2010 4th IEEE International Conference on Digital Ecosystems and Technologies (DEST), Dubai, United Arab Emirates, 13–16 April 2010; pp. 256–261.
35. Reinisch, C.; Kofler, M.J.; Iglesias, F.; Kastner, W. ThinkHome energy efficiency in future smart homes. *EURASIP J. Embed. Syst.* **2011**, *2011*, 1–18.
36. The Saving Energy in Europe's Public Buildings Using ICT (SMARTSPACES) Project. Available online: http://www.smartspaces.eu/ (accessed on 31 October 2014).
37. The OSGi Alliance. OSGi Release 5. Available online: http://www.osgi.org/Specifications (accessed on 31 October 2014).
38. Fensel, A.; Tomic, S.; Kumar, V.; Stefanovic, M.; Aleshin, S.V.; Novikov, D.O. SESAME-S: Semantic smart home system for energy efficiency. *Inform. Spektrum* **2013**, *36*, 46–57.
39. Fensel, A.V.; Kumar, V.; Tomic, S. End user interfaces for energy efficient semantically-snabled smart homes. *Energy Effic.* **2014**, *7*, 655–675.
40. Kumar, V.; Fensel, A.; Fröhlich, P. Context Based Adaptation of Semantic Rules in Smart Buildings. In Proceedings of the International Conference on Information Integration and Web-Based Applications & Services, Bali, Indonesia, 3–5 December 2012.
41. Kumar, V.; Fensel, A.; Lazendic, G.; Lehner, U. Semantic Policy-Based Data Management for Energy Efficient Smart Buildings. In On the Move to Meaningful Internet Systems: OTM 2012 Workshops, Rome, Italy, 10–14 September 2012; Herrero, P., Panetto, H., Meersman, R., Dillon, T., Eds.; Springer: Berlin-Heidelberg, Germany, 2012; Volume 7567, pp. 272–281.
42. Sang, J.; Ye, C.; Hu, H.; Li, R.; Fu, L.; Yang, D.; Xiang, H.; Fu, C. Semantic web-based policy interaction detection method with rules in smart home for detecting interactions among user policies. *IET Commun.* **2011**, *5*, 2451–2460.
43. Girtelschmid, S.; Steinbauer, M.; Kumar, V.; Fensel, A.; Kotsis, G. On the application of Big Data in future large scale intelligent Smart City installations. *Int. J. Pervas. Comput. Commun.* **2014**, *10*, 168–182.
44. Gruber, T. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum. Comput. Stud.* **1995**, *43*, 907–928.
45. DogOnt Ontology Documentation. Available online: http://elite.polito.it/ontologies/dogont/dogont.html (accessed on 31 October 2014).
46. DogOnt Ontology OWL download. Available online: http://elite.polito.it/ontologies/dogont.owl (accessed on 31 October 2014).
47. Furfari, F.; Sommaruga, L.; Soria, C.; Fresco, R. DomoML: The Definition of a Standard Markup for Interoperability of Human Home Interactions. In Proceedings of the 2nd European Union Symposium on Ambient Intelligence, Eindhoven, The Netherlands, 8–10 November 2004; pp. 41–44.
48. The Dog Gateway Project. Available online: http://dog-gateway.github.io (accessed on 31 October 2014).

49. Bonino, D.; Corno, F. SpChains: A declarative framework for data stream processing in pervasive applications. *Procedia Comput. Sci.* **2012**, *10*, 316–323.