Emulating Home Automation Installations through Component-based Web Technology

J. A. Asensio**, J. Criado*, N. Padilla*, L. Iribarne*

Applied Computing Group, University of Almería, Spain

Abstract

The Internet of Things mechanisms enable the management of home environments since they can be developed as IoT based information systems. From standard smart homes to automated buildings, including other kind of domotics and inmotics solutions, every system must be tested and validated before its installation. The current tools offered by IoT and home automation vendors lack in emulation features close to the real behavior of the devices. In many cases, delaying the verification actions until the hardware is acquired and installed may cause some drawbacks, for example, from the economic point of view. This paper presents a solution for emulating home automation environments which are based on the KNX standard and can be represented by architectures of devices. The emulation consist of developing virtual implementations of real devices which operate and communicate through web technology. The technology implementing these virtual devices allows us to develop components which can provide different type of data related to the installation (audio, video, text, animations, images, etc.). The architectures can be managed using web services and their behavior can be tested through web user interfaces showing the mentioned data. Furthermore, virtual and physical devices are connected to validate the interoperability between the real installation and the emulation.

Keywords: Home Automation, Emulation, KNX, Multimedia Web Components, IoT, COScore infrastructure

1. Introduction

Regarding all the environments that surround us (including familiar, professional and social facets), appliances and devices are increasingly used to help in resolving tasks, offer us some useful information or capture data from the context, among other possible examples. In this sense, smart installations are becoming normal in houses, buildings and even entire cities. In most situations, devices conforming these installations are (in some way) responsive to the presence and behavior of agents interacting with them, meeting the Ambient Intelligence (AmI) and Internet of Things (IoT) domains [1]. Within this context, the communication between agents and the installations can be performed using different types of interaction (*e.g.*, through a user interface, gestures, or voice inputs) which can cause different behaviors as a result. Furthermore, previous domains intend to enable the customization and configuration of their device networks and they may depend on many variables (such as user profiles, context aware information, system capabilities, etc.) which can be taken into account in the development or not, thus causing the reconfiguration must be performed at run-time. As a consequence, smart environments should be flexible for adapting their behavior to the requirements and the ambient conditions, which can be different in each scenario.

One of the main related issues of this kind of hardware-dependent systems is that they cannot modify their physical structure whenever new scenarios need to be solved. Therefore, developers and installers must take into account all the possible situations included in (or derived by) the requirements to build the final solution. Moreover, the required tasks experience the additional difficulty of not having a wide variety of tools for the life cycle management. For example,

*Corresponding author

**Principal corresponding author

Email addresses: jacortes@ual.es (J. A. Asensio), javi.criado@ual.es (J. Criado), npadilla@ual.es (N. Padilla), luis.iribarne@ual.es (L. Iribarne)

some design tools are dependent on the vendors or the selected technology. Even some devices from companies like Nest only provide user interfaces (via web or through a mobile application) to monitor them and change their functioning options, but do not provide any software to configure the relations between devices for example, for establishing rules of interrelated behavior. In the case of KNX technology [2], ETS Inside includes this kind of reconfiguration software, but only for KNX devices and restricting the possible operations to simple parameterization actions over the devices which are present in the installation. In many cases, developers cannot test and validate the constructed system until the real devices are installed in the corresponding facility, unless they have a test laboratory providing all the required devices in addition to the wiring.

From our perspective, this kind of IoT based cyber physical systems require some tools for managing a complete product life cycle, including the stages of analysis, design, development, test and maintenance. However, most of the related tools available in the market are focused on design and initial configuration capabilities, without providing enough functionality to perform a system adaptation. In some cases, this is determined by the low flexibility of the hardware. For example, some home automation devices are parameterized at design-time before their installation and cannot change their behavior at run-time. As a consequence, a re-parameterization of the affected devices, or the physical modification of the installation (*e.g.*, replacing or reconnecting devices) is necessary, in the case that new requirements must be fulfilled.

On the one hand, this characteristic emphasizes the non-dynamic nature of smart environments and encourages the need of performing complete tests before the sensors, actuators and rest of the elements are acquired and installed. On the other hand, it could be interesting to have a flexible operating mode that enables some kind of adaptation by using the same available physical devices but varying the previous system behavior. In this sense, it may be possible to adapt the communication process following a perspective of architectures made upon devices, representing the feasible communications by the connections among them and modifying these relations to obtain a new behavior. For this reason, it would be necessary a mechanism for managing these architectures built from home automation devices. Moreover, this mechanism should provide the required operations to enable the execution of the constructed architectures and the adaptation of their structure and behavior, for example, to perform changes in the installations.

This paper propose a specific mechanism for emulating home environments using the web technology for (1) developing an implementation of each real device and (2) performing the communication processes (see Figure 1). In particular, the presented solution is focused on home automation installations based on the KNX standard and

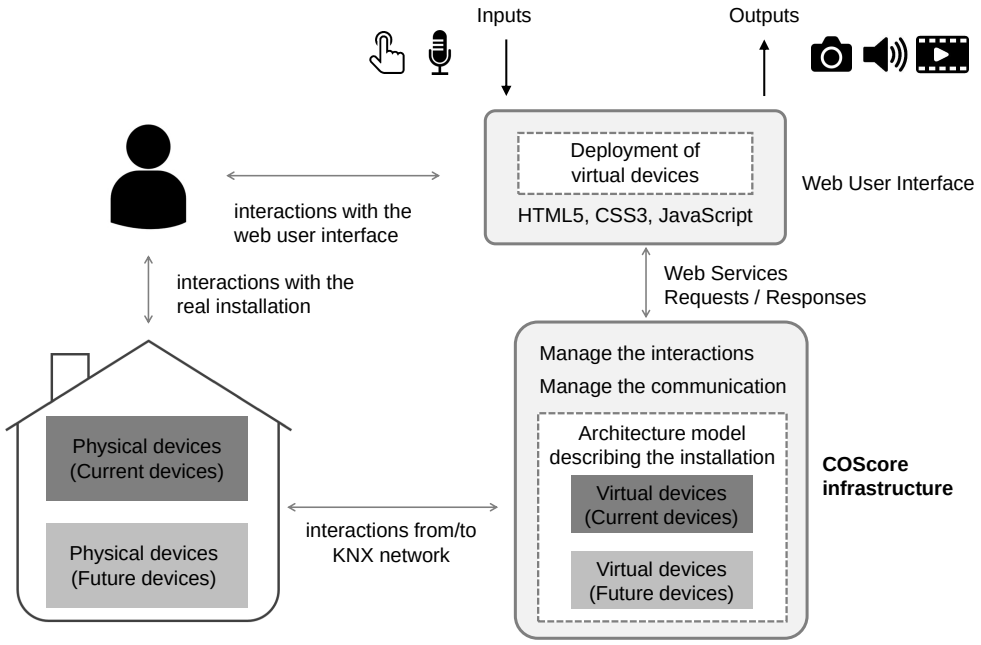


Figure 1: Schema of the approach for emulating home automation installation

which structure can be represented by architectures of devices communicating each other. These architectures are used for describing the physical devices of a real installation but can also contain additional devices corresponding to elements which are not yet present and may be part of a possible extension (both types are called virtual devices). In addition, the architectures can include other software components intended to manage the installation but which have no correspondence with real devices. The components of the architectures are managed using an infrastructure called COScore developed in a previous research work [3]. This infrastructure is based on web services and implemented under the REST (REpresentational State Transfer) principles. It has been applied as a management backend to allow the emulation of behavior of home automation installations. Furthermore, in the current proposal, virtual and physical devices are connected to validate the interoperability between the real installation and the emulation.

Moreover, the behavior of the emulated installation can be tested and validated through simple web user interfaces which provide the capability of interacting with the devices and observing the outputs and behavior generated. Since the capabilities of home automation systems include different media of interaction, our approach must provide the support of multimedia communication and the type of components which can (1) obtain different data types as an input (*e.g.*, voice commands and clicks on the graphical user interface) and (2) provide different data types as an output (*e.g.*, streaming from cameras, audio notifications, images and animations of emulated behaviors). For example, a virtual device corresponding to an audio sensor can be developed to get voice commands from users and parse the indicated actions to a comprehensible language. In addition, we are able to represent the state of a roller shutter through an animation showing the movement of raising and lowering of an virtual representation. As a consequence, we need that technologies implementing such components support the management of different media. In this sense, user interfaces relying on Web technology offers the required features to acquire as inputs and offer as outputs different data types such as images, videos, animations, text or audio, among other possibilities.

Our approach can be useful for evaluating home automation installations before the hardware is acquired, which involves a lower risk when a development is carried out. In addition, test and validation operations can be accomplished since the interaction performed in virtual devices have an impact in real ones, and vice versa, and the obtained results are shown to the user. Furthermore, the generated emulation artifacts can be used by developers and installers as prototypes to show the behavior of the installations thus assisting the meeting with clients and contractors.

The rest of the paper is structured as follows. Section 2 describes the background infrastructure based on web services to manage software architectures. Section 3 presents our perspective of a home automation installation, including design and implementation aspects necessary to describe our approach. Section 4 demonstrates and validates the application of the approach by emulating the behavior of an example scenario. Section 5 reviews the most relevant related works. Finally, Section 6 draws conclusions and proposes the future work.

2. Background infrastructure based on SaaS

As described in the previous section, to build an application to emulate a home automation installation, it is necessary to use a technology infrastructure that we have built in a previous research work. This infrastructure, called COScore, is used for the deployment of COTS component-based architectures and its development is based on three fundamental pillars: Component-based Software Engineering (CBSE), Model-Driven Engineering (MDE) and Cloud Computing.

Component-based Software Engineering [4] impacts on the software development by reusing elements and parts (*i.e.*, components) thus improving the reliability and productivity since the time required for creating such software can be reduced. This discipline is focused on integrating software components into an application following a bottom-up development instead of a traditional top-down perspective. In our case, each home automation application is defined as a set of components (forming an architecture) which are obtained from one or more third-party repositories. These components are called COTSgets, from Commercial Off-The-Shelf (COTS) [5] and gadgets (understood as any software that can work alone or as a piece of the architecture). A COTS component is any coarse-grained component developed by third parties available for building more complex systems. In the next subsections, the component COTSgets and the architectures based on this type of component are described in more details.

Model-Driven Engineering [6] is focused on building models on different levels of abstraction which improves the specification and definition of software applications. This discipline also provides different mechanisms to facilitate the automation or semi-automation of software development. For example, model transformation techniques [7] can be applied to generate different levels of software specifications, even generating the code of a final product.

MDE techniques can assist some tasks related to component-based software systems, for example, the architectural design and development by describing the structure and behavior, or the functional and non-functional properties [8], among other possible examples. Our component-based architectures are represented on three levels of abstraction: (1) Abstract architectural model, corresponding to the Platform Independent Model (PIM) level in Model Driven Architecture (MDA) [9], and representing the architecture in terms of the type of components it contains and their relationships; (2) Concrete architectural model, corresponding to the Platform Specific Model (PSM) level, and describing the concrete components that comply with the definition of the abstract architecture; and (3) Final software architecture, which represents the source code (our components) to be executed or interpreted. These architectures are managed by the COScore infrastructure and provided to the clients for their deployment, as show in Figure 2. It is noteworthy to remark that this figure shows an example of management over a mashup user interface [3]. In the current approach, the COScore is used to manage complete home automation installations, including graphical components for user interfaces and non-graphical components for the rest of purposes (*e.g.*, controllers, actuators, etc.), as part of the same software architecture.

Cloud Computing paradigm [10, 11] provides a set of benefits for users, organizations and applications, including the use of *Software-as-a-Service* (SaaS). An specific kind of software is a model representing it (or a part) in a high level of abstraction and therefore the concept of *Models-as-a-Service* (MaaS) can be applied to identify the use and request of model resources on demand. The combined use of MaaS and MDE can improve the availability, run-time sharing, scalability and distribution of models [12]. Our approach focuses on the management of software architectures based on COTSget components, as a specific use of MaaS.

The result is a cloud service which offers *COTSgets-as-a-Service* and the operations required to ensure the capabilities of our component-based architectures. It therefore includes: (a) management of the COTSgets specifications, (b) management of the COTSgets-based architectures, (c) instantiation of COTSgets components, (d) initialization of user applications based on the architectures, and (e) communication of components belonging to an architecture. All these capabilities are offered at run-time to dynamically provide architecture and component models, thus using concept of *COTSgets-as-a-Service*. Furthermore, this service makes the main parts (such as the databases of components and architectures, the platform-dependent server or the platform-independent server) highly scalable and distributable as additional benefits derived from the cloud computing [11]. Moreover, the purpose of this service is to support interactive systems running on different platforms. Nevertheless, the current version of this cloud service only supports the management of component-based applications on the web platform. The following subsection describes the languages proposed (using an MDE perspective) for describing COTSget-based architectures.

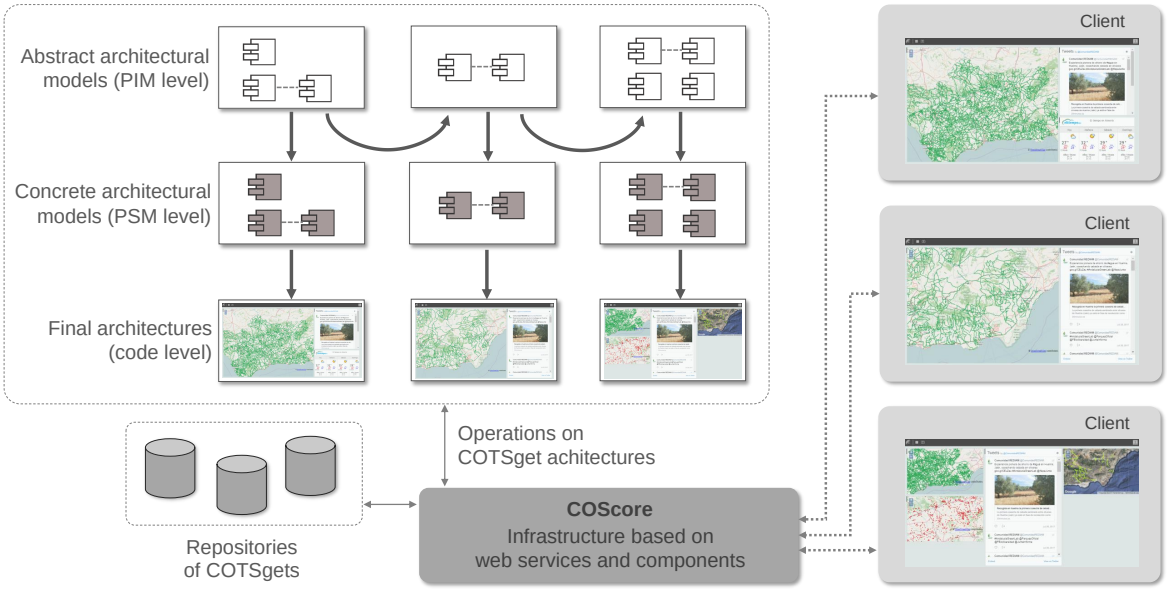


Figure 2: COScore infrastructure [3] to manage component-based architectures

2.1. COTSget-based architectures

A Domain-Specific Language (DSL) has been proposed to describe our COTSget-based architectures, which is shown in the metamodel diagram of Figure 3. This metamodel defines the abstract syntax of our DSL using the Meta-Object Facility (MOF) as our metamodeling language [13, 14]. As previously stated, our architectures are represented in a abstract level (*Abstract Architectural Model*) and a concrete level (*Concrete Architectural Model*). The former identifies the abstract components, *i.e.*, the types of components an architecture must include to be considered correct. The latter identifies the concrete components that have been selected as a solution for the types defined in the abstract architectural models. Both architectural models define the relationships established among components.

Each component has a type, which can be *container*, *functional*, *userInteration* or *normal*. A *container* identifies a component containing other components. This makes it possible to build complex components from basic ones. A

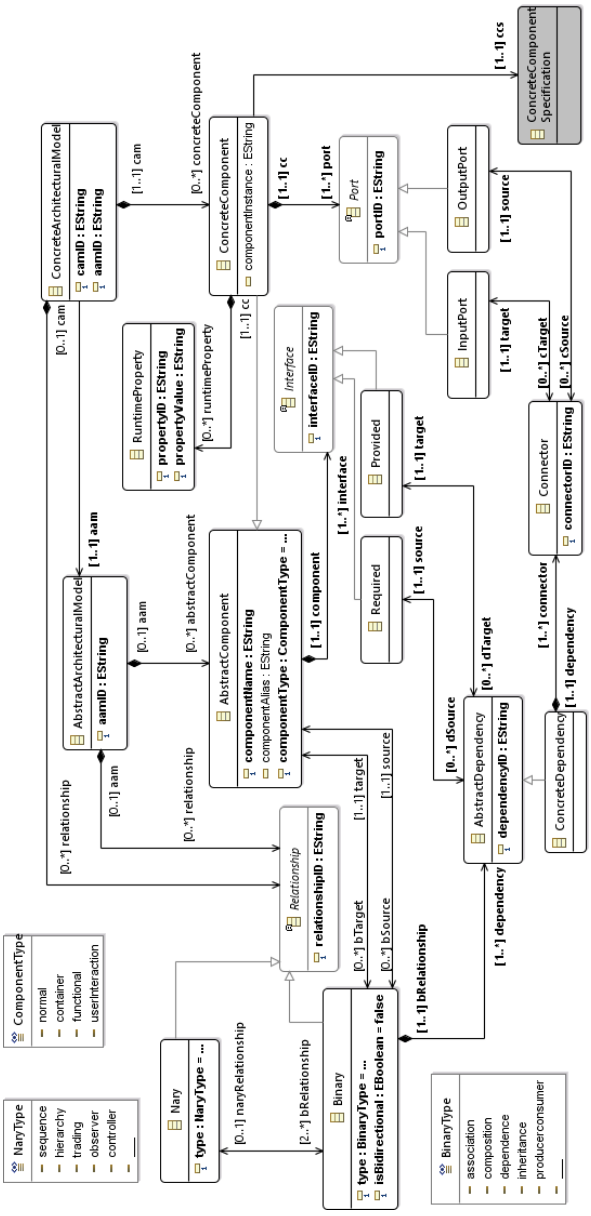


Figure 3: Metamodel of our component-based architectures

functional component does not include user interaction, and therefore, is built to execute background code (internal code of the component). A *userInteraction* component includes the management of user interaction or provides information to the user. A *normal* component is the union of *functional* and *userInteraction* component types.

In the case of concrete components, these elements contain a list of the properties that have been modified or that should be taken into account at run-time. Each component has a reference to its corresponding specification and it is discussed in the following subsection. Moreover, a component has a list of ports which are responsible for communication between components. A relationship between concrete components is formed by connectors, which represent the links that allows us to send information from the output ports to the input ports. Relationships can be binary or n-ary. Binary relationships are established between two different components (*e.g.* association, composition, etc.), whereas n-ary relationships are composed of at least two binary relationships and are therefore related to at least three components (*e.g.* hierarchy, sequence, etc.).

This language allows us to formally describe the structure of our component-based architectures. Such architectures must be initialized from existing applications or constructed by developers performing an abstraction process. In any case, the applications (which are being executed on the client side) and the architectural models (which are being managed on the server side) must always be synchronized. Therefore, if the cloud service changes (proactively) the architecture (adding new components, removing unnecessary elements, etc.), the changes in the new model are propagated to the client side. Furthermore, if the client modifies the architecture, the cloud service executes (reactively) the corresponding synchronization (see Figure 2).

2.2. COTSgets components

The specifications of each concrete component, referenced from the language described in Figure 3 through the element *ConcreteComponentSpecification*, are defined by the language described by the metamodel of Figure 4. Each component is composed of four parts: functional, extra-functional, packaging and marketing. Figure 4 shows only the functional and extra-functional parts since both comprise the essential information to (1) describe a component and, in particular, (2) apply this concept to proposed approach for emulating home automation installations. The marketing part of a component identifies the information related to the entity which has developed the component, such as the name of the organization, contact name, etc. The packaging part provides information identifying the repository where the component is located, the programming language, etc. The extra-functional part describes these properties may provide information on non-functional attributes (NFPs), quality of service (QoS), component appearance and layout, and any dependencies on other components.

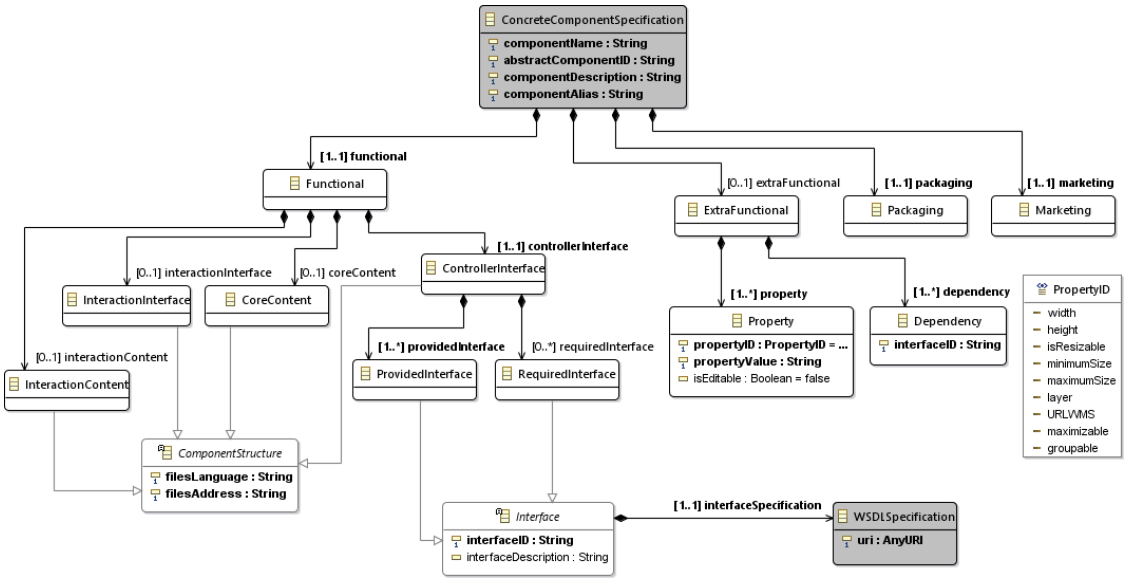


Figure 4: Metamodel of our components

Regarding the functional part, a component includes the interaction content implementing the business logic of the interaction, and the *core content* which is the main part to store the methods implementing the rest of a component business logic. In addition, a component includes two interfaces, the *interaction interface* storing the methods for handling interaction events, and the *controller interface* which is the part in charge of handling the requests and responses derived from the use of the functional interfaces. In our approach, the operations of a component are accessible from the outside through the interaction and controller interfaces. Once any part receives a method call, it can transfer the execution flow to the component parts with which it is related. The interaction interface is related to the controller interface and the interaction content. The controller interface is connected to all the component parts. The interaction content is related to the core content (see Figure 5). The controller interface is used for communication between components. It is comprised of a set of provided and required interfaces, with the constraint that each component must have at least one provided interface. The provided interfaces define the visible functionality, *i.e.*, methods that can be invoked to make the component perform some operation. The required interfaces describe the operations belonging to other components that must be invoked to work properly.

Provided and required interfaces are defined with the structure of the Web Services Description Language (WSDL). The port type concept of WSDL 1.1 [15] is used as the root element for describing each interface. Each interface has an identifier, so each interface in the same component can be referred to unequivocally, and a set of operations. An operation must always have an input and, optionally an output. The input may be formed by a set of elements (parameters) while the output part consists of one or more elements (returned types). More information about the definitions of COTSgets and architectures can be found in [3].

Following this component specification, the functionality implemented in a component must be defined in the *interaction content* and *core content*. The methods of these parts must invoke the related *controller interface* operations if the business logic is intended to exchange information with other components. Therefore, the data that can be obtained from a component is the information provided through its ports, with are contained in the functional interfaces of the *controller interface* (see Figure 5).

3. Design and Implementation of Home Automation Installations

The foundation for understanding the solution proposed in this article lies in our perspective of smart environments in general, and home automation installations in particular. Therefore, this section describes how we address their development, focusing on design and implementation aspects.

As previously stated, we follow a CBSE perspective to represent all the devices of an automated home. As a consequence, a home automation network is described by an *architecture model* of devices connected via their interfaces and, consequently, their ports. If two devices are not connected, they will not able to send/receive data to/from the other one. Due to our goal is to perform emulation task, *physical devices* have an equivalent implementation in software, thus conforming the *virtual devices*. Such elements must include (1) a business logic implementation that emulates the real behavior and (2) the interfaces needed to enable the communication with the rest of devices.

In these architectures, sensors, actuators, controllers are represented as COTSgets components which can only be accessed and managed through their functional interfaces. Additionally, extra-functional data completes a *device specification*, including properties (*e.g.*, dimensions, average response time, etc.), packaging (*e.g.*, location, implementation language, etc.) and marketing (*e.g.*, price, vendor, contact address, etc.) information.

Figure 6 shows the specification of a generic switch controller device. With regard to the functionality, it has three provided and two required interfaces. The first type allows the execution of operations whereas the second type is used to make requests to other devices. The calls to interfaces' methods (see the description of Figure 6a) are solved

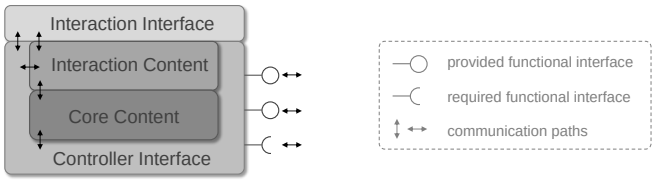
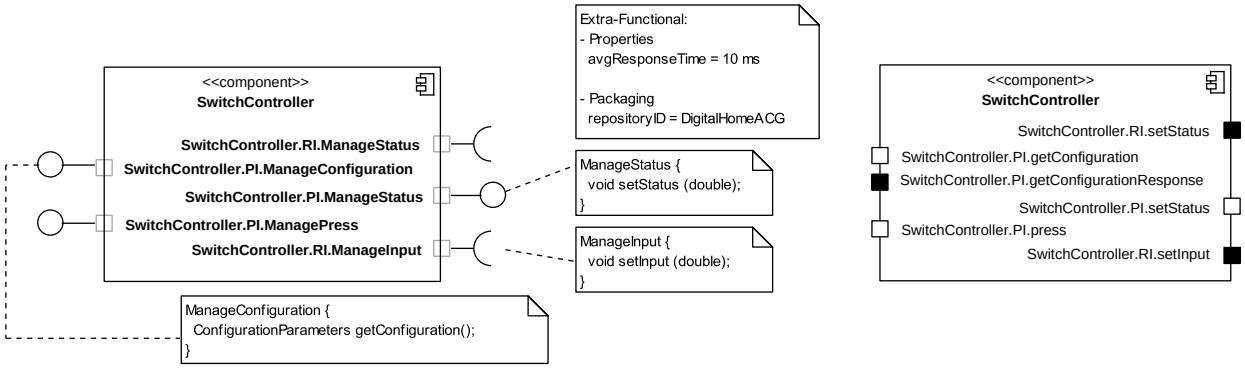


Figure 5: Component parts



(a) Description of interfaces

(b) Description of ports

Figure 6: Example of a device specification

by using input and output ports (see Figure 6b). Depending on the type of interface and if the method returns a value, the correspondence to ports differs.

For example, the method *setStatus* of the provided interface *ManageStatus* is in charge of modifying the status value of the component. It accepts a parameter typed as double and has no return value, therefore, it is implemented with an input port. As another example, the method *setInput* of the required interface *ManageInput* (which is responsible for changing the input value of an actuator) accepts a parameter typed as double and has no return value but, in this case, the method is solved by using only one output port (because the required interface indicates the external use of this functionality). Furthermore, when methods have a return value, two ports are necessary (as in the case of the *getConfiguration* method from the provided interface *ManageConfiguration*). When the interface is provided, the execution of a method starts in its input port and ends in the output port. On the contrary, if the interface is required, the execution starts in the output port and ends in the input one.

From the aforementioned design of home automation devices, we propose to implement them through web technology. This characteristic allows us to perform emulation, test and validation operations related to an installation from any location with the only requirement to have an Internet connection.

The virtual devices are implemented as WebComponents[1] using Polymer[2] and therefore, component parts are made up of HTML5, CSS3 and JavaScript files. This technology enables (*i*) the encapsulation of a device implementation, (*ii*) the use of HTML templates to develop components following the device specification structure, (*iii*) the reuse of devices by applying different parameters and therefore deploying distinct execution properties, and (*iv*) the acquisition and deployment (*i.e.*, management) of different data types such as text, images, video, audio, etc.

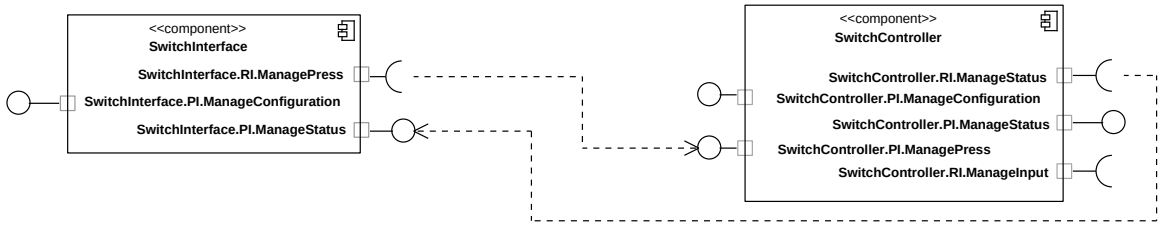
In addition, these files will be structured as shown in Figure 5 organized in four folder, one for each component part. Focusing on the interaction interface, event handlers and listeners form part of the JavaScript implementation code. With regard to the controller interface, the calls to functional interfaces' methods are solved by means of communication ports implemented as WebSockets [16] with the Socket.IO[3] library. The components deployed on the client side are managed and communicated using the COScore infrastructure as intermediary [3]. Figure 7 shows an example of two devices connected and ready to communicate each other.

In this example, the *SwitchInterface* is a virtual device in charge of providing a simple graphical user interface to change the status of a switch. It has been connected to the aforementioned *SwitchController*. Both devices are interrelated through *ManagePress* and *ManageStatus* interfaces. On the one hand, the switch interface requires the use of *ManagePress* operations to emulate the behavior of pressing the switch. Consequently, it sends the notification to the switch controller. On the other hand, the controller requires the use of the methods from *ManageStatus* interface to inform about the switch status. Then, it sends this data to the switch interface as needed.

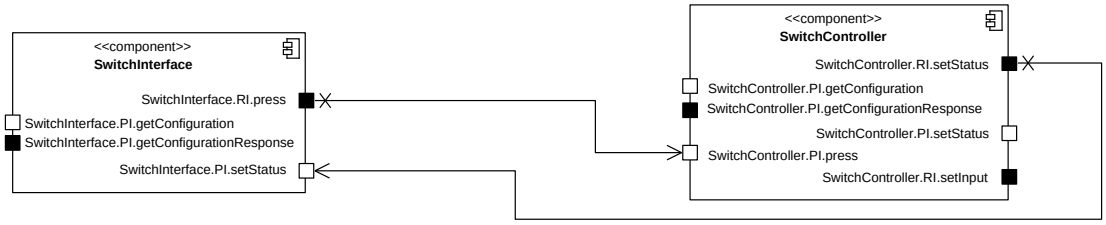
[1] WebComponents – `http://webcomponents.org/`

[2] Polymer Project – `https://www.polymer-project.org/`

[3] Socket.IO – `http://socket.io/`



(a) Description of interfaces



(b) Description of ports

Figure 7: Communication between two virtual components

In our approach, virtual and physical devices are interconnected with the goal that interaction performed in the emulated elements is also received by the physical devices, and vice versa. Thus, the behavior of a home automation installation can be evaluated and tested by analyzing the results from two sources, and the interoperability between the real installation and the emulation can be validated.

Regarding the relationship shown in Figure 7, when a user emulates the press action by interacting with the switch interface, a message is sent to the physical network through a gateway and the output port *press* is used for communicating with the *SwitchController*. The message consists of a telegram that complies with the KNX protocol whose destination is the adequate group address. This fact implies that our Polymer components implementing the virtual devices must be initialized with the target group address corresponding to the physical elements. Furthermore, when a physical device sends a telegram, it must be captured by the gateway to transmit the data to the corresponding virtual element.

The communication with the KNX network can be performed through the web protocol if the installation is equipped with a web server, or an IP gateway. Another option is to establish a USB connection if this kind of interface is used. Depending on this communication, the gateway is implemented with different components. Nevertheless, the messages to the input ports are sent by using WebSockets and the telegrams to the physical network are sent by using a Node.js client for EIB/KNX installations [17].

Our approach proposes to represent only the virtual devices in the architecture models. They can implement the behavior of home automation or non home automation devices (see the right side of Figure 8). The names of the components implementing these devices start with the initial of the specific technology or with NHA (Non Home Automation), respectively. Furthermore, there are also other components in the installations which have no correspondence with a physical device (left side of Figure 8). The last elements are intended to implement a business logic different from the behavior of physical devices, but they may be useful for evaluation purposes or the deployment of graphical user interfaces on different platforms. This type of component always needs a related component acting as it interface to be managed (*e.g.*, a graphical user interface of a button for representing a switch). Nevertheless, components with a correspondence with a physical device only need a linked interface component when they are being emulated and there are no physical interface to manage them (dashed squares in Figure 8).

4. Emulating the Behavior of an Example Scenario

With the aim of validating the feasibility and benefits of the approach presented in this paper, an example susceptible to appear in a real situation is presented below. This example scenario comes from an existing home automation system intended to control the light of a room. This installation is implemented with KNX technology and consists

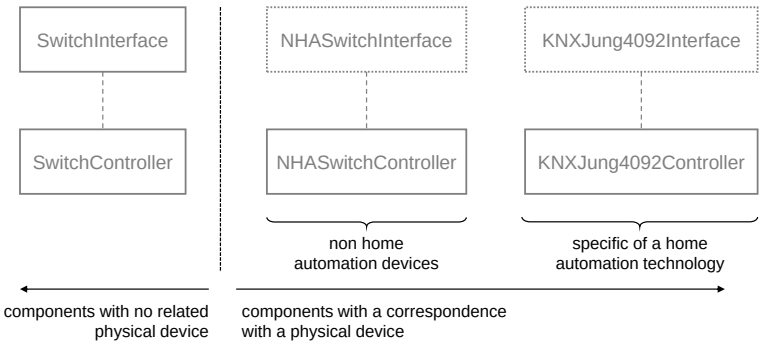
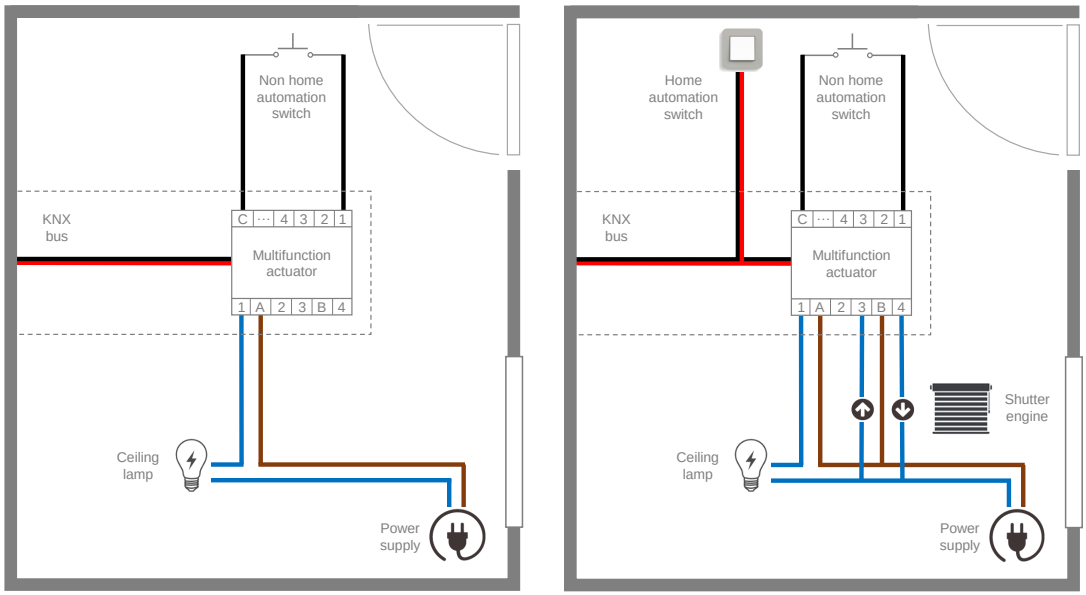


Figure 8: Different types of components in our home automation installations

of: (a) a KNX power supply; (b) a multifunction hybrid actuator with six inputs and four outputs, which is directly connected to the bus; (c) a ceiling lamp; and (d) a non home automation switch to turn the light on and off. The lamp and switch are connected to the actuator using the appropriate wiring. Figure 9a shows a schematic representation of the starting scenario.

The goal is to extend this installation for including the control of a roller shutter located in the same room. To do this, an engine that allows raise or lower it by interacting with a home automation switch will be used. This switch is composed by two buttons at it will work as follows. A long press on the button located on the top will cause the shutter to rise until it reaches its limit whereas a long press on the button located on the bottom will produce the shutter to lower until it reaches its stop. If a short press on either of the two buttons is done, the engine will stop the movement of the shutter. Finally, if the long press of a button is performed and then, the other button is also activated with a long press, the shutter switches its movement after a period of security to avoid damaging the engine. Therefore, the target installation can be represented as shown in Figure 9b.

Before making the investment in the devices and elements needed to carry out the expansion, our proposal would analyze the feasibility and study the best configuration for the new home automation installation. For this purpose,



(a) Initial installation

(b) Target installation

Figure 9: Home automation installations of the example scenario

the first step is to make a model of the new system, where both the existing elements in the installation and the new elements to incorporate are represented, in addition to the connections between them. Figure 10 shows the model describing the COTSget-based architecture of the system focusing on the description of functional interfaces. There is a controller for each physical device (*NHASwitchController*, *KNXZennioACTinBOXClassicHybridController* (actuator) and *NHALampController*) and also for each emulated device (*KNXABBSwitchController* and *NHAShutterController*). The components *SwitchInterface*, *KNXABBSwitchInterface*, *LampInterface* and *NHAShutterInterface* represent the switches, lamp and shutter, respectively, in a web user interface. The element *AudioInterface* corresponds to a component in charge of receiving voice commands as an input in addition to play the audio related to the notifications of the actions performed in the home automation installation. As mentioned before, the latter components cannot be directly related to a controller of a physical device. For this reason, the components *SwitchController*, *LampController*, *KNXABBSwitchController*, *NHAShutterController* and *AudioController* are necessary.

This model shows that controllers of existing physical devices has no related component intended to the web interface deployment (*e.g.*, *NHASwitchController*). On the contrary, controllers of emulated elements have this type of component related to them (*e.g.*, *NHAShutterController*). Consequently, existing physical devices can be managed by interacting with them or by using additional software components, *e.g.*, *SwitchInterface* for the switch. In this case, the *SwitchController* is required because the *SwitchInterface* cannot be connected directly to the *NHASwitchController*. This means, for example, the lamp will be turned on and off whether acting on the switch through the web interface as if the located physical button is pressed. In addition, the change in the corresponding device status is reflected in the web UI. The same behavior happens by acting on the shutter button, except that both the push button and the shutter engine mechanism is emulated (showing a graphical representation in the web interface). The component *KNXZennioUSBController* matches a physical device operating as a gateway between all the components and the real physical devices.

With regard to connections between components, Figure 10 includes the relations through the required and provided interfaces. Next, Subsection 4.1 describes the communication process and the connection between virtual and physical devices by using an example. The, Subsection 4.2 analyze in detail the implementation of one of the main components from the example scenario, the *KNXZennioACTinBOXClassicHybridController*. The rest of components are implemented following the same guidelines, each of them with its specific functionality.

4.1. Testing the Execution from the UI and the Real Devices

Due to the existence of both physical and virtual devices in our home automation system, the following communication flows can be established: (a) communication between physical devices, (b) communication between virtual devices (*i.e.*, components), and (c) communication between physical devices and components. It is noteworthy to remark that a software component of the architecture describing the home automation system can be related to (*i*) a virtual home automation device which has a correspondence in the real world with a physical device, (*ii*) a virtual non home automation device which also has an equivalent physical device, or (*iii*) a virtual device with no equivalence with a physical element. The three types of interaction between devices are exemplified below through a real situation: turn on the light in the room.

In the proposed scenario, when the user interacts with the web interface with the aim of turning on the lamp, the following sequence is carried out (see Figure 11): (1) the user activates the push button of the *SwitchInterface*; (2) the component *SwitchInterface* uses its required interface *ManagePress* to communicate with the provided interface of the component *SwitchController*; (3) the last one uses its required interface *ManageInput* to communicate with the provided interface *ManageInputs* of the component *KNXZennioACTinBOXClassicHybridController* indicating the input status change; (4) in order to send the command to the physical device, the component *KNXZennioACTinBOXClassicHybridController* communicates through its required interface *ManageMessages* with the corresponding provided interface of the *KNXZennioUSBController*, which generates the telegram and injects it into the KNX bus; (5) this telegram is interpreted by the target physical device (the actuator) and it activates the output for turning on the physical lamp and also sends a telegram with the new status; (6) the telegram with the status is received by the *KNXZennioUSBController* and it uses the required interface *ManageMessages* to communicate with the *KNXZennioACTinBOXClassicHybridController* to notify the change; (7) the actuator uses its required interface *ManageStatus* for sending the information to the *NHASwitchController*, *SwitchController*, *NHALampController* and *LampController*, which update their data with the new status; (8) finally, the *SwitchController* and *LampController* use their required interface *Man-*

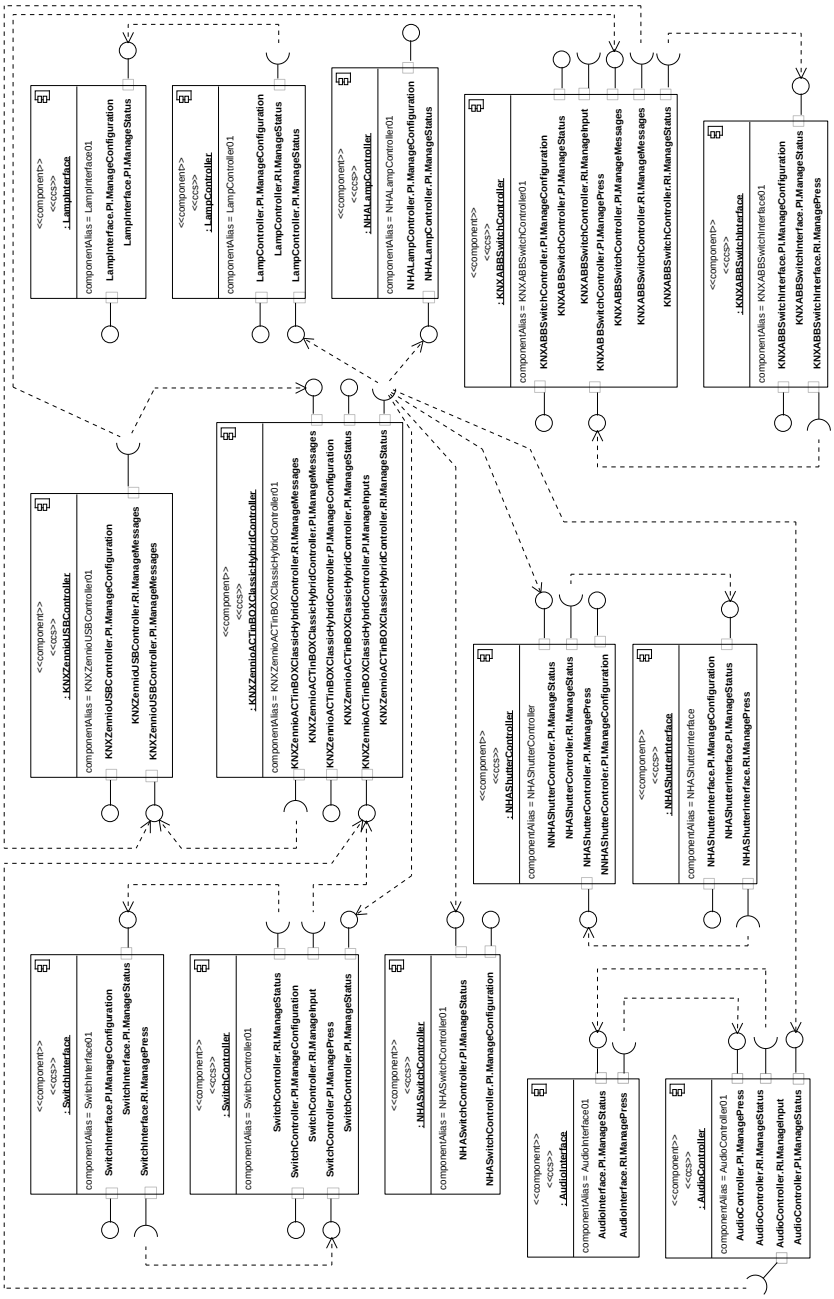


Figure 10: Concrete Architecture Model of the installation

ageStatus to communicate with the components *SwitchInterface* and *LampInterface*, respectively, thus updating the information shown in the web interface, *i.e.*, showing the lamp has been switched on.

In the case of any other interaction over the system is performed, a similar process is carried out. The established communication flows highlight the usefulness of the proposal to verify the configuration of the devices in a home automation installation and emulate their behavior. It allows us to confirm the requirements with a customer and offer a web application that enables the interaction with the physical installation from devices external to the installation such as a PC, tablet, smartphone, etc.

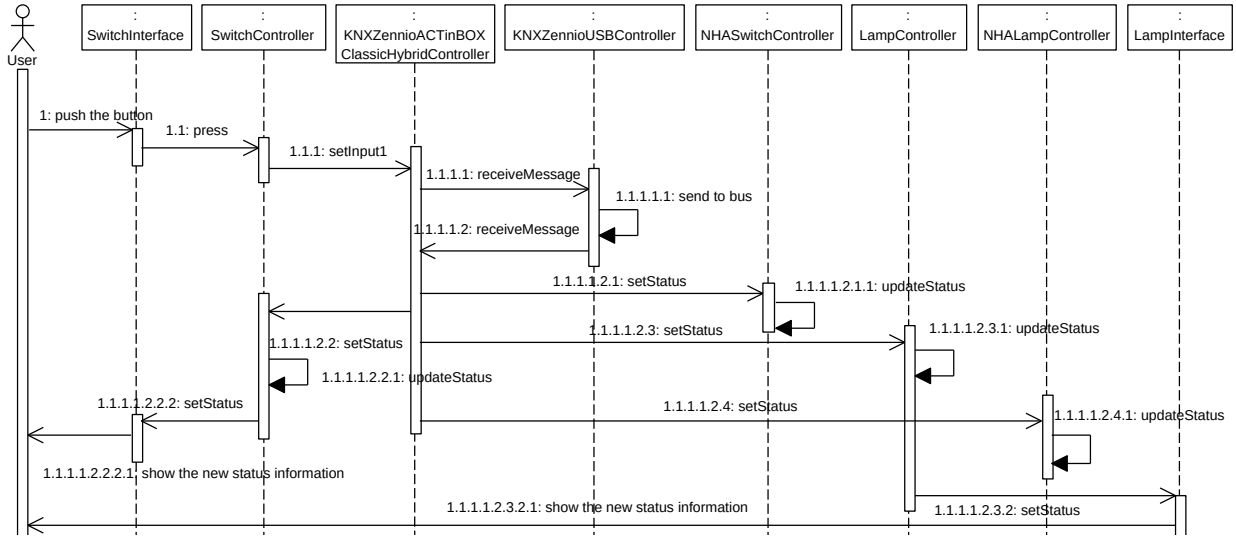
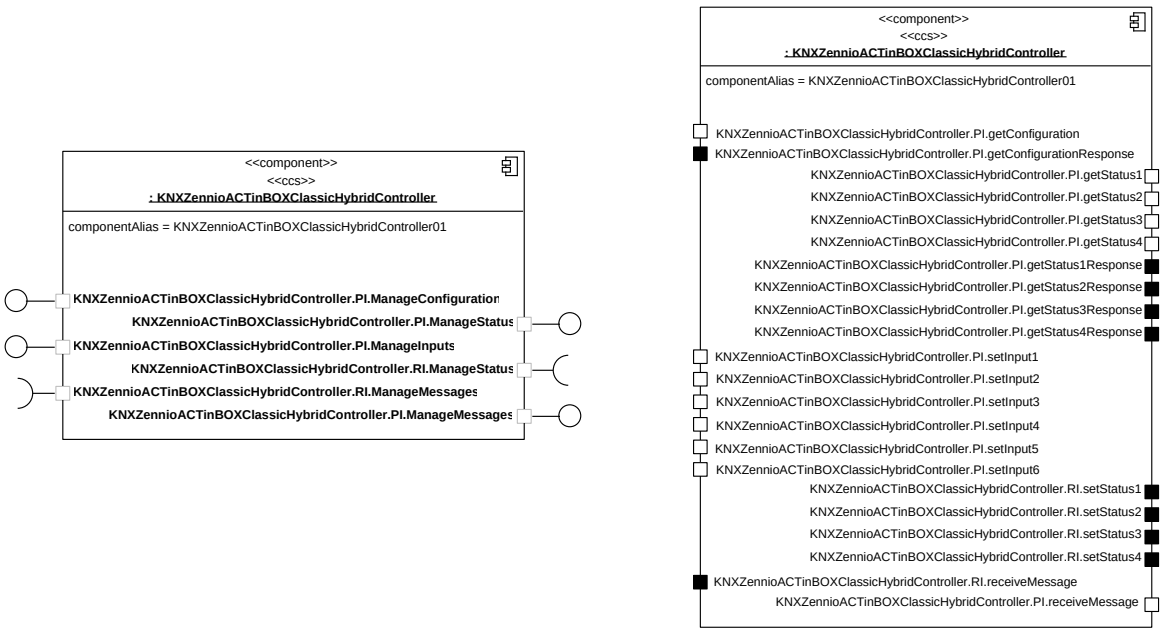


Figure 11: Sequence diagram describing the action of turn on the lamp

4.2. Example Device: Multifunction Hybrid Actuator

As mentioned in Section 3, the components implementing the virtual devices and utilized in the scenario has been developed following the guidelines of W3C for WebComponents, and we have used the Polymer library, applying HTML5, CSS3 and JavaScript technologies. Next, we describe in some detail the characteristics of the implementation of one of the most important components of the installation, *KNXZennioACTinBOXClassicHybridController*.

The device Zennio ACTinBOX Classic is a multifunction actuator with six inputs and four outputs including logical operations, which results in a wide variety of possible configurations. In our model, the *KNXZennioACTinBOX-*



(a) Interface view of the component

(b) Port view of the component

Figure 12: Specification of the actuator component

ClassicHybridController represents this device and implements its main features. Figure 12 shows its specification from the interface and port views. This component has four provided interfaces (*ManageConfiguration*, *ManageStatus*, *ManageInputs* and *ManageMessages*) and two required interfaces (*ManageStatus* and *ManageMessages*) if we focus on the left side of the figure. On the right side, the ports forming part of each interface are shown. The interface *ManageConfiguration* allows us to modify its configuration; for this reason, it has an input port (*getConfiguration*) to receive the request and an output port (*getConfigurationResponse*) to return the information about the setting. The provided interfaces *ManageInputs* and *ManageStatus* enable the management of inputs and outputs, respectively. The required interface *ManageMessages* is in charge of communicating with the *KNXZennioUSBController* to send telegrams to the KNX bus, whereas the provided interface with the same name deals with the reception from the USB controller. The required interface *ManageStatus* is responsible for sending the output status to the components related to the actuator. The provided interface with the same name is intended to read the status of the outputs managed by the actuator.

The structure of this component is summarized in Figure 13. The starting point is the *index.html* file and it links the rest of the artifacts containing the implementation of the different component parts (described in Subsection 2.2). This file contains the component definition within the *dom-module* label. The style sheet must be located in the beginning of the element and the content to be shown to the user is defined within the *template* label. Due to the actuator is a component without a graphical interface, both are empty. Next, all the necessary artifacts containing the implementation of each of the four parts (*interaction interface*, *interaction content*, *controller interface* and *core content*) defined in the component metamodel are linked. This file also contains the component creation and registration. Finally, other required libraries necessary for the correct execution are included. In this case, the *socket.io* library is used for the communication with the rest of components through the COScore infrastructure, as described in Section 2.

The rest of the section contains the most relevant details about the assets referenced from the *index.html* file. In the case of the interaction (*/content/interaction/interaction.js* and */interface/interaction/interaction.js*), the resources are empty because this component has no user interface and it does not have to implement any business logic related to such interaction. The section does not describe the file *connection.js* since it contains all the functions enabling the communication with the COScore infrastructure and it is not the goal of this article.

Asset Component (component.js)

This asset refers to file that contains the definition of an object with the component's identifier, its properties and the methods implementing its life cycle. The component identifier is specified using the attribute *is* of the object. The component's public *properties* (those which can be set and modified) are also stated in this file. The names given

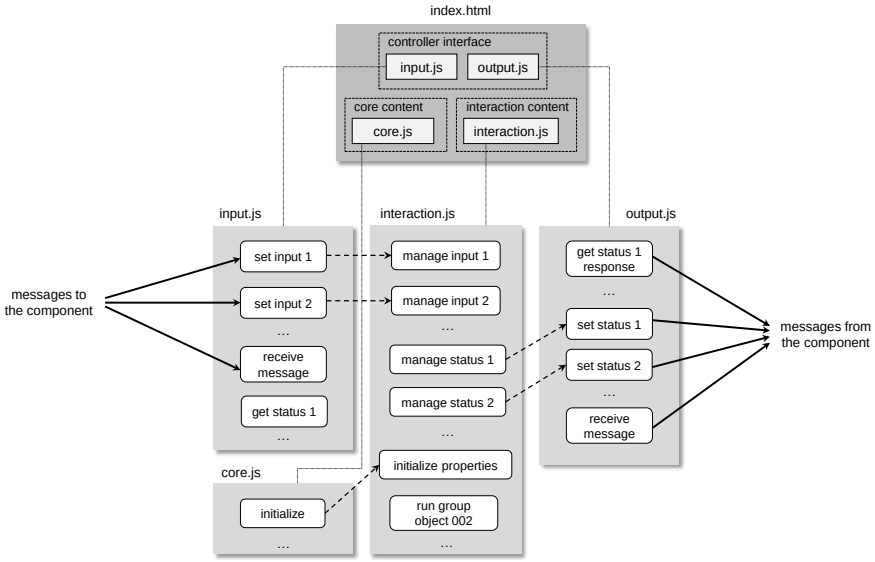


Figure 13: Structure of the actuator component implementation

to the properties of the example component are: (a) *user_id*, identifier of the user interacting with the application; (b) *nodejs_url*, address of the JavaScript server of the infrastructure which manages the communications; (c) *component_name*, type of the component; (d) *component_alias*, descriptive name to differentiate components of the same type; and (e) *component_instance*, unique identifier of the component instantiation.

The component defines an additional attribute called *dataStore* to keep all the information related to the physical device. It includes a *dictionary* to store the values of the labels for each configurable parameter of the device; the internal *properties* of the device (*e.g.*, the model or the individual address); the *parameters* that can be modified in the device (active inputs, default output values, or time settings, among other examples related to this component); the *groupObjects* which describe characteristics about the type of data (*Datapoint Type*) of value represented, the identifier of the input or output concerned (if applicable), etc. The *dataStore* also contains the connection established between each group object and the ports of the component (*objectPortLinks*), the relation between the previous and the group addresses (*objectPortAddressLinks*), timers, internal variables (for example, if the component is set or not in emulation mode), etc.

At the end of the object definition, the function *ready* implements the state (within the component life cycle) that is executed once the values of their public properties have been established and the local DOM (Document Object Model) has been created and initialized. In this state, the *dataStore* properties are initialized and the connection with the COScore infrastructure is established.

Asset Core (content/core/core.js)

The file related to this asset consists of a series of functions implementing all the component's business logic. The function *initialize* performs a number of more specific function calls where the dictionary, internal properties, parameters, group objects, etc., are initialized. More details about the communication in KNX networks depending on group objects and group addresses can be found in [18]. Within the function *initializeGroupObjects*, the following data is defined: the data type (*Datapoint Type*) of the represented value is specified for each group object, the input identifier, the output or channel concerned, and if it is related to a short or long press (where it is applicable). The function *initializeObjectPortLinks* connects each group object to the related port with its the corresponding interface, although there may also be unused group objects.

The file *core.js* also contains the common getters and setters to obtain and modify the properties' values. Moreover, there are other relevant functions such as *evaluateInput* which is in charge of obtaining the new input status from the configured press value ('Sending of 0/1', 'Shutter Control', etc.), the response ('0', '1', 'Switching 0/1', 'Up', 'Down', 'Stop', etc.), and occasionally the old input status. In a similar manner, the function *evaluateOutput* obtains the new output status from the output type ('Normally Open', 'Normally Closed', 'Shutter (No slats)', etc.) and the input status ('0' or '1').

Finally, the functions *runGroupObject[number]* implement the business logic of each group object. This methods can be triggered either by receiving the corresponding telegram from the bus, or if the associated port of such object is activated, and their functionality vary depending on whether the component is set or not in emulation mode. The emulation mode is used when the physical device is not installed in the home automation system (as the component *KNXABBSwitchController* of the example scenario) and therefore, all its functionality is emulated, including the generation and sending of telegrams to the bus (through another specific component that operates as the gateway, the *KNXZennioUSBController*).

In the case that emulation mode is not activated for the actuator component, it works as a transmitter of telegrams to other components through its interfaces. The use of the emulation mode in this component has a number of implications for the type (physical or virtual) of sensors and actuators that can be connected to its inputs and outputs. Table 1 lists all the possible combinations that may occur. Next, we discuss some of the functions *runGroupObject[number]* as an example of this scenarios.

Regarding the inputs, the functions *runGroupObject066* and *runGroupObject084* are related to short and long press actions of input number 1, respectively, and their goal is to obtain the new input status and store it. With regard to the outputs, if the output number 1 of the component is activated, the function *runGroupObject002* is in charge of managing the communication (with the related components) derived from this change through the port *setStatus1* from the required interface *ManageStatus*. In addition, if the emulation mode of the actuator is activated, it executes the functionality of the group object 006, which is responsible for responding the emulated call to the input port *getStatus1*, thus sending the new output status through the port *getStatus1Response* from the provided interface

ManageStatus. In this last case, it makes no sense to send the telegram to the bus as it is not possible to connect a physical device to a component output.

Asset Input (interface/controller/input.js)
This asset corresponds to the file *input.js*, which contains the methods implementing the component's input ports belonging to the required and provided interfaces. As mentioned in Section 3, communication between ports has been implemented using WebSockets with *socket.io*. It enables the possibility to be listening (input ports) and broadcast (output ports) events. In the case of the input ports, it has been implemented the function *registerInputs* to define their listening events, where the identifiers of each port are recorded, establishing the link with their specific functions.

For example, the function *setInput1* receives from other virtual device (*i.e.*, other component) the activation of the input number 1 and, taking into account the type of press performed (short or long), it infers the related group address. Then, it sends a telegram to this address with the corresponding group object (in this case, the object 066) and the new input value. To this end, the method *setInput1* calls the reception message function which corresponds to the output port *receiveMessage* from the required interface *ManageMessages* (this port is implemented in *output.js*). Moreover, the *input.js* artifact also defines the *receiveMessage* function which handles the reception of messages that have been sent to the bus and then it calls the corresponding function *runGroupObject[number]* of the core asset.

Asset Output (interface/controller/output.js)
Finally, the file *output.js* implements the methods corresponding to the component's output ports, such as *getStatus1Response*, *setStatus1* or *receiveMessage*. The first two functions only emit an event to establish the communication with the connected input ports, sending the status of the output number 1 and the activation/deactivation of such output, respectively. Regarding the output port *receiveMessage*, it is in charge of sending the telegrams that must be injected to the KNX bus through the component which is acting as the gateway (*i.e.*, *KNXZennioUSBController*) between virtual and physical devices.

As complementary material, we provide a web page[4] which includes some code fragments of the aforementioned files. Thus, more details about the implementation are offered with the aim to better understand the proposed approach of using the Polymer web technology to emulate home automation instalations.

4.3. Discussion about the results obtained in the example scenario

The result obtained from designing and implementing the example scenario following the specifications of Section 3 is an architecture of COTSgets components that can be managed by our COScore infrastructure. Once the model describing this architecture is incorporated to the system, the components corresponding to the home automation installation can be deployed on the web client thus offering a user interface intended to manage and monitor the devices. As depicted in Figure 9 and modeled in Figure 10, the installation is basically composed by a light and a roller shutter that can be managed from different inputs. The light is present in the current physical installation whereas the shutter is a device to be possibly incorporated in the future. In the architecture, both devices can be managed by interacting with (*i*) the NHA and KNX switches of the real installation, (*ii*) the buttons of the KNX switch virtual representation deployed in the web user interface, and (*iii*) the web UI through voice commands. As a consequence of this interaction, the results of the actions can be observed both in the physical installation (in the case of light) and in the web user interface. In this UI the results are (1) shown graphically with and image (for light) and an animation (for the roller shutter) and (2) emitted as audio data, thus reporting the performed action.

Emulation	In. type	Input example	Out. type	Output example
Deactivated	Physical	Real switch	Physical	Real lamp
	Physical	Real switch	Virtual	WebComponent lamp
	Virtual	WebComponent switch	Physical	Real lamp
	Virtual	WebComponent switch	Virtual	WebComponent lamp
Activated	Virtual	WebComponent switch	Virtual	WebComponent lamp

Table 1: Possible scenarios

[4]Web page with complemetary material – `http://acg.ual.es/emuhai/`

As shown in the Figure 14, the UI is intended to be an equivalent representation of the real installation. It deploys the components of the example architecture (Figure 10), although only the interface type components are accessible, *i.e.*, *KNXABBSwitchInterface* (a), *AudioInterface* (b), *LampInterface* (c) and *NHAShutterInterface* (d). It is important to note that in this UI, the *SwitchInterface* of the model has been deactivated, since we try to focus on the validation and evaluation of the KNX switch implementation. The testbed utilized in this case study (left side of Figure 14) is a briefcase gathering the elements of a real home automation installation and therefore it includes more physical devices than those modeled in our example architecture. The installation is connected to a Raspberry Pi through a KNX USB interface device. The Raspberry Pi accesses to Internet and communicates with the Node.js server of our COScore infrastructure, thus acting as a gateway between our approach and the KNX network. This way, virtual and physical devices are interconnected and both (1) the interoperability between them and (2) the emulation of home automation devices has bee validated.

The complementary material of the web page includes a video with a demo execution of the proposed scenario. In this case, the light is activated by using a voice command ('Action 1'). Then, the physical lamp is switched on and its virtual representation in the web user interface is also updated (showing an image of a lamp on). Additionally, the user receives an audio notification (it occurs repeatedly for each action carried out in the installation). Next, the light is deactivated through the voice command ('Action 2'). As a consequence, the physical lamp is switched off and the UI shows an image of a lamp off. Then, the user clicks on the right-bottom part of the KNX switch, thus starting the downward movement of the roller shutter (represented with an animation in the UI). Next, the same part of the button is pressed to stop it. The right-up part of the switch starts and stops the upward movement. The left part of the switch is pressed to commutate the state of the light, as an alternative to use the voice commands to switch on/off the lamp.

5. Related Work

Previous research work in the literature deals with the management of existing home automation installations describing the technologies and communication mechanism from a low level perspective [19]. In our case, we address the control of these systems as a main aspect of the emulation process and from a high level point of view trying to isolate the remote access mechanism by using web sockets. In this sense, the control and construction of home automation systems may deal with integration issues in order to allow communication between devices from different manufacturers or operating with different protocols [20]. This fact leads to the emergence of solutions and tools based on high-level languages and web technology to enable the management of existing installations and to facilitate the design and development of new systems [21].

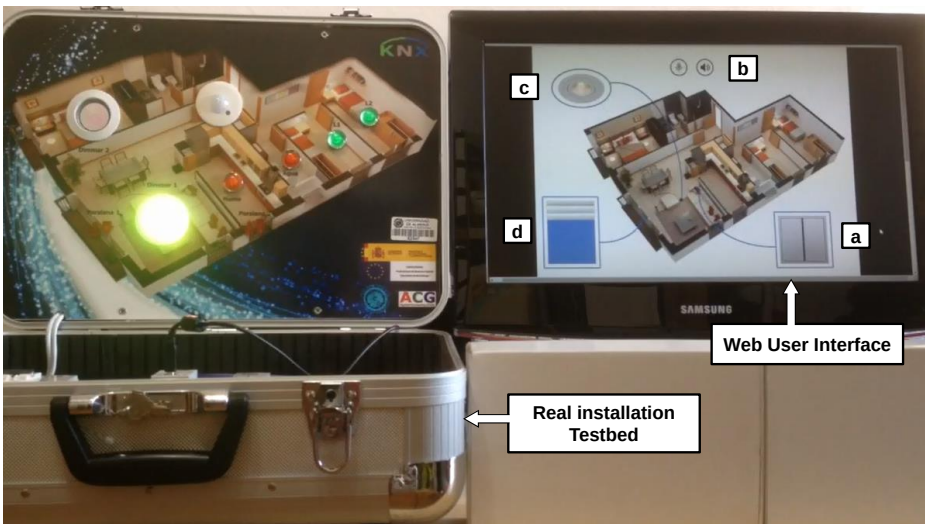


Figure 14: Web user interface provided for emulating the scenario

Regarding simulation capabilities (*i.e.*, only virtual components are deployed in the architecture and no real devices are present), the DogSim approach [22] describes the devices' behaviors through state machines and this proposal provides an SCXML compliant simulation engine for injecting new events and reacting to events and state changes of these simulated machines. Based on this previous work, the authors of EmuDog [23] integrates the DogSim features and the Dog domotic gateway [24] for mixed environments in which events can be simulated or caused by real device interactions. EmuDog approach discovers devices needing emulation from the house model and builds the corresponding state machine. All machines (from virtual or real devices) are taken into account to receive new events and possibly generate other ones as a consequence of changing their states. In contrast, our emulation proposal does not require the previous phase of building state machines for managing event injections. The components' interface specifications (and their implementation) are enough to execute the devices' behaviors. In addition, our approach is deployed on web technology and all the components (real and emulated) can be managed invoking RESTful services.

The approach presented in [25] describes oBIX (open Building Information Exchange), a standard based on XML and web services intended to represent and access smart home and building automation data. The paper is focused on the mapping between KNX and oBIX representations, and how it provides the means to construct a remote management station to control, monitor and change parameters of an installation. The use of web technology to offer a homogeneous application layer for KNX networks have also been addressed in [26]. That work is focused on the implementation of a gateway enabling the access to devices of a KNX network by using RESTful web services. The main difference to our work is that both approaches (oBIX and the gateway) are intended to provide a REST API from an existing ETS project and the services generated are focused on getting information from devices and performing read/write operations with the network's group objects. In contrast, our approach is intended to develop applications with emulation capabilities by executing the devices' behavior and showing the generated result in a web user interface. In addition, the communications to the KNX network in [26] and [25] are performed from the server side by using Calimero [27], instead of using Javascript calls in virtual device implementations from the client side, as we do.

Other type of intermediate (apart from gateways) is proposed in [28], where authors describe an object-based middleware called OSHNet which supports heterogeneous communication protocols and messages by applying a new definition of communication objects at the network layer. In our case, we do not deal with different protocols and we are focused on solving the communication processed derived from the emulation of home automation installations.

There are in the literature previous attempts of facilitating the development of applications to manage KNX devices. In [29], the authors describes an open SDK (Software Development Kit) in C language to build KNX-based systems from building scripts, source code and configuration files. Apart from the Javascript library [17] used in this paper for implementing the communication among devices, there are other possibilities of libraries such as [30], which is based the previous library, or [31], which is inspired in a C# API [32] intended for managing KNX installations. This kind of libraries is used in our to approach to enable the communication with devices from this technology. In the case that physical devices from other proprietary technology are managed with our approach, the corresponding libraries must be used in the implementation of the virtual devices.

The are complete frameworks such as the openHAB (open Home Automation Bus) [33], OpenRemote [34], pimatic [35] or Freedomotic [36] which provide a vendor and technology agnostic open source software for developing home automation systems. In addition, these platforms offer some mechanisms to allow the reconfiguration of the architectures and the modification of devices' behaviors. However, these frameworks cannot emulate the behavior of devices which are not present in the physical installation neither enable the communication between both types of devices (*i.e.*, physical and virtual).

6. Conclusions and Future Work

The approach presented in this paper is oriented to facilitate the design, implementation and maintenance of home automation installations made up of two different type of devices: physical and virtual. The first type refers to the already known hardware devices and the second type identifies the software components used to manage the physical devices or involved somehow in the interaction with the installation performed by users. For example, a virtual device can be the implementation of a component which executes the same behavior of a real physical element. Moreover, virtual devices can also be graphical components deployed in a web page intended to manage a installation, for example, a button to open the garage door or a display to show if a specific light is switched on.

In this research work, we focused on the virtual devices of this kind of cyber physical systems with the goal of developing web applications which emulate the behavior of home automation installations. The resulting software allows us to enable the user interaction not only through the physical elements but also using components, for example, through a web user interface. The web user interfaces include components managing different types of data, enabling the multimedia communication and the interaction with audio commands, animations emulating the real behavior, buttons, or simple text reporting the current status, among other possible examples. Furthermore, by applying the business logic implemented in the emulated installation, it is possible to observe and analyze the results derived from the interaction and the configuration of the devices. These results can be displayed in a web user interface but they can be also observed in the physical installation, because virtual devices are connected to the physical network through a gateway. In addition, due to this gateway, the execution derived interaction performed over the physical elements is also propagated to the virtual components.

The approach has been addressed by using a technology infrastructure called COScore based on CBSE, MDE and Web Services (it is described in previous research work [3]). Within this infrastructure, home automation installations are described through models containing the specifications of the architectures and each particular component. Therefore, components are managed as black-boxes and can be accessed and managed through their functional interfaces. This encapsulation mechanism has been translated to the implementation phase with the use of WebComponents specifications and the Polymer library. As a consequence, the web applications which emulates the home automation installations are entirely developed with HTML5, CSS3 and JavaScript technologies. In addition, the communications between components are implemented with web sockets through a JavaScript server which also allows the operation between the client (web applications) and the server (COScore back-end). In any case, each component can be accessed independently through the invocation of its communication ports thus following an IoT approach. This specification of component and its implementation using Polymer can establish a common reference to create a community of developers of software components which enables the emulation of home automation devices and facilitates the development of home automation installations.

The integration between physical and virtual media allows us to evaluate the behavior of existing installations but it also enables the possibility of deploying virtual devices for testing possible expansions of the systems without the need of acquiring the physical devices. Moreover, testing and validation operations related to the evaluation process may assist the development of home automation systems (either at early stages, or subsequent maintenance tasks). The components and the web applications also facilitate the communication between clients and installers or contractors. Furthermore, the architectures which are used for deploying the home installations can be modified within the infrastructure to adapt their behavior to new ambient conditions.

The presented paper assumes that a software engineering is in charge of building the model which represents the home automation installation. Therefore, our intention is to develop a mechanism which helps us in the construction of such model, as a future work. On the one hand, we want to implement a tool for enabling the graphical edition of the models. On the other hand, we intend to get the model of the installation (which is already in the PSM level) from its CIM definition following an MDA approach by executing an automatic or semi-automatic process carried out by model transformations. We want to incorporate a set of robust security mechanisms to solve the potential risks associated to home automation installations [37]. Moreover, we want to validate our approach in different home automation technologies apart from KNX. Thus, our proposal of emulating of software architectures could form part of certain types of smart cities ecosystems. We also will investigate about the feasibility of performing re-parameterization of physical devices at run-time with the aim of enabling the re-configuration not only of components but also the modification of the business logic of the physical installation.

Acknowledgements

This work was funded by the Spanish Ministry MINECO under Project TIN2013-41576-R.

References

[1] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): A vision, architectural elements, and future directions, Future Generation Computer Systems, 29(7) (2013) 1645–1660. doi: 10.1016/j.future.2013.01.010.

[2] KNX Standard. ISO/IEC 14543-3. https://www.knx.org/, 2006 (accessed 05.04.17).

[3] J. Vallecillos, J. Criado, N. Padilla, L. Iribarne, A cloud service for COTS component-based architectures, Computer Standards & Interfaces 48 (2016) 192–216. doi: 10.1016/j.csi.2015.11.008.
[4] I. Crnkovic, M. Larsson, Challenges of component-based development, Journal of Systems and Software 61(3) (2002) 201–212. doi: 10.1016/S0164-1212(01)00148-0.
[5] L. Iribarne, J.M. Troya, A. Vallecillo, A trading service for COTS components, The Computer Journal 47(3) (2004) 342–357. doi: 10.1093/comjnl/47.3.342.
[6] M. Brambilla, J. Cabot, M. Wimmer. Model-driven software engineering in practice. Synthesis Lectures on Software Engineering. Morgan & Claypoll (2012). doi: 10.2200/S00441ED1V01Y201208SWE001.
[7] T. Mens, P. Van Gorp. A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152 (2006) 125–142. doi: 10.1016/j.entcs.2005.10.021.
[8] I. Crnkovic, S. Sentilles, A. Vulgarakis, M. Chaudron, A classification framework for software component models, IEEE Transactions on Software Engineering 37(5) (2011) 593–615. doi: 10.1109/TSE.2010.83.
[9] A.G. Kleppe, J.B. Warmer, W. Bast, The model driven architecture: practice and promise, Addison-Wesley Professional, 2003.
[10] C.A. Lee, A perspective on scientific cloud computing, In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ACM (2010) 451–459.
[11] M. Whaiduzzaman, M. Nazmul Haque, M. Rejaul Karim Chowdhury, A. Gani, A Study on Strategic Provisioning of Cloud Computing Services, The Scientific World Journal Volume 2014, Article ID 894362 (2014) 16 pages. doi: 10.1155/2014/894362.
[12] H. Brunelire, J. Cabot, F. Jouault, Combining model-driven engineering and cloud computing, In Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10: Workshop's 4th edition (2010). https://hal.archives-ouvertes.fr/hal-00539168/.
[13] ISO/IEC 19508, Information Technology – Object Management Group – Meta Object Facility (MOF) Core (2014).
[14] J. Sánchez Cuadrado, J. García Molina, Building domain-specific languages for model-driven development. IEEE Software 24(5) (2007) 48–55. doi: 10.1109/MS.2007.135.
[15] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1, W3C, March 2001. http://www.w3c.org/TR/wsdl, 2001 (accessed 05.04.17).
[16] V. Wang, F. Salim, P. Moskovits, The definitive guide to HTML5 WebSocket (Vol. 1), Apress, Berkeley, Calif, USA, 2013.
[17] node-eibd. A Node.js client for eib/knx daemon. https://github.com/andreek/node-eibd, 2016 (accessed 05.04.17).
[18] KNX Communication. http://www.knx.org/fileadmin/template/documents/downloads_support_menu/KNX_tutor_seminar_page/basic_documentation/Communication_E1212a.pdf (accessed 09.04.17).
[19] P. Belimpasakis, V. Stirbu, A survey of techniques for remote access to home networks and resources, Multimedia tools and applications 70(3) (2014) 1899–1939. doi: 10.1007/s11042-012-1221-y.
[20] M. Jung, J. Weidinger, W. Kastner, A. Olivieri, Building automation and smart cities: An integration approach based on a service-oriented architecture, In 27th International Conference on Advanced Information Networking and Applications Workshops (WAINA), IEEE (2013) 1361–1367.
[21] A. Zanella, N. Bui, A. Castellani, L. Vangelista, M. Zorzi, Internet of things for smart cities, IEEE Internet of Things Journal 1(1) (2014) 22–32. doi: 10.1109/JIOT.2014.2306328.
[22] D. Bonino, F. Corno, Dogsim: A state chart simulator for domotic environments, In 8th IEEE International Conference on Pervasive Computing and Communications Workshop, IEEE (2010) 208–213.
[23] D. Bonino, F. Corno, Modeling, simulation and emulation of intelligent domotic environments, Automation in Construction 20(7) (2011) 967–981. doi: 10.1016/j.autcon.2011.03.014.
[24] D. Bonino, E. Castellina, F. Corno, The DOG gateway: enabling ontology-based intelligent domotic environments, IEEE transactions on consumer electronics 54(4) (2008) 1656–1664. doi: 10.1109/TCE.2008.4711217.
[25] M. Neugschwandtner, G. Neugschwandtner, W. Kastner, Web services in building automation: Mapping knx to obix, In 2007 5th IEEE International Conference on Industrial Informatic, Vol. 1, IEEE (2007) 87–92.
[26] G. Bovet, J. Hennebert, A web-of-things gateway for knx networks, In Proceedings of 2013 European Conference on Smart Objects, Systems and Technologies (SmartSysTech), VDE (2013)1–8.
[27] The Calimero Project, A free KNX network library. http://calimero-project.github.io/, 2016 (accessed 05.04.17).
[28] S.O. Park, J.S. Kim, S.J. Kim, An object-based middleware supporting efficient interoperability on a smart home network, Multimedia tools and applications 63(1) (2013) 227–246. doi: 10.1007/s11042-011-0926-7.
[29] W. Kastner, G. Neugschwandtner, Kögler M. An open approach to eib/knx software development, IFAC Proceedings Volumes 38(2) (2005) 255–262. doi: 10.3182/20051114-2-MX-3901.00035.
[30] node-red-contrib-eibd, KNX/eib nodes for node-red. https://github.com/ekarak/node-red-contrib-eibd, 2016 (accessed 05.04.17).
[31] knx.js. https://github.com/estbeetoo/knx.js, 2016 (accessed 05.04.17).
[32] knx.net, KNX API for .NET. http://lifeemotions.github.io/knx.net/, 2016 (accessed 05.04.17).
[33] OpenHAB, empowering the smart home. http://www.openhab.org/, 2016 (accessed 05.04.17).
[34] OpenRemote, Open Source Middleware for the Internet of Things. http://www.openremote.com/, 2016 (accessed 01.08.17).
[35] pimatic, smart home automation for the raspberry pi. https://pimatic.org/, 2017 (accessed 01.08.17).
[36] Freedomotic, Open IoT Framework. http://freedomotic.com/, 2017 (accessed 01.08.17).
[37] A. Jacobsson, M. Boldt, B. Carlsson, A risk analysis of a smart home automation system, Future Generation Computer Systems, 56 (2016) 719–733. doi: 10.1016/j.future.2015.09.003.

José Andrés Asensio

José Andrés Asensio is PhD in Computer Science at the University of Almería since 2013. He joined the Applied Computing Group (TIC-211) in 2007, a research group at the University of Almería. He has participated as a researcher and collaborator in national research projects (refs. TIN2006-06698, TIN2007-61497, TRA2009-0309, TIN2010-15588, and TIN2013-41576-R) and a regional research project (ref. P10-TIC-6114) since 2006. During the academic year 2008/09, he was half-time Associate Professor at the University of Almería. Since 2003/04, he has been professor and/or coordinator of several courses and workshops. He currently works as a Technician in the Foundation of the University of Almería. His research interests include: Model-Driven Engineering, Ontology-Driven Engineering, Component-Based Software Engineering, Trading, Software Agents, Multi-Agent Systems, Home Automation, and Digital Home.

Javier Criado

Javier Criado is a PhD in CS at the University of Almería. He received the Computer Science Engineering degree and the Master degree in Advanced Computer Techniques from the University of Almería, Spain. In 2009, he joined the Applied Computing Group (TIC-211), a research group in Computer Science from the University of Almería. Since then, he has participated in three national research projects (refs. TIN2007-61497, TIN2010-15588 and TIN2013-41576-R) and a regional research project (ref. P10-TIC6114). Since 2011, he is supported by an FPU grant (ref. AP2010-3259). His research interests include: Model-Driven Engineering, Component-Based Software Engineering, Model Transformations, Mashups, Model-Driven Development for Advanced User Interfaces, COTS components, Trading, Agents and Multi-Agent Systems, Ontology-Driven Engineering and UML design.

Nicolas Padilla

Nicolas Padilla received the B.S. and Ph.D. degrees in computer science from the University of Almería in Spain, and the M.S. degree in computer science from the University of Granada, Spain. From 1991 to 1993, he worked as an Associate Professor at the University of Granada. Since 1993, he has worked as Associate Professor in the Advanced College of Engineering at the University of Almería. In 2007, he co-founded the Applied Computing Group. His research interests include cooperative systems, model-driven engineering, human-computer interaction, agents and multi-agent systems, and UML design.

Luis Iribarne

Luis Iribarne is a Lecturer at the Department of Informatics, University of Almeria (Spain). He received the BS and MS degrees in Computer Science from the University of Granada, and the PhD degree in Computer Science from the University of Almeria, and conducted from the University of Malaga (Spain). From 1991 to 1993, he worked as a Lecturer at the University of Granada, and collaborated as IT Service Analyst at the University School of Almeria. Since 1993, he has served as a Lecturer in the Advanced College of Engineering at the University of Almeria. From 1993 to 1999, he worked in several national and international research projects on distributed simulation and geographic information systems. Since 2006, he has served as the main-coordinator of five R&D projects founded by the Spanish Ministry of Science and Technology, and the Andalusian Ministry ST. In 2007, he has founded the Applied Computing Group (ACG). He has also acted as evaluator for funding agencies in Spain and

Argentina. He has published in referred JCR scientific international journals (ISO Abbrev): Comput. J., Comput. Stand. Interfaces, J. Log. Algebr. Methods Program, Softw.-Pract. Exp., Simul. Model. Pract. Theory, IEEE Trans. Geosci. Remote, J. Neurosci. Methods, Inf. Syst. Manage., Behav. Brain Res., Comput. Ind., IEEE Trans. Syst. Man Cybern. Paart A-Syst. Hum., J. Vis. Lang. Comput. (among others). He has also published in referred scientific international conferences (ICMT, ICSOC, ICSOFT, SOFSEM, ICAART, PAAMS, SEAA, EUROMICRO, among others) and book chapters. His main research interests include simulation & modeling, model-driven engineering, machine learning, and software technologies and engineering.