Modeling, Simulation and Emulation of Intelligent Domotic Environments

Modeling, Simulation and Emulation of Intelligent Domotic Environments

Dario Bonino[a], Fulvio Corno[*,a]

[a]*Politecnico di Torino, Dipartimento di Automatica ed Informatica, Corso Duca degli Abruzzi 24, 10129 - Torino, Italy*

Abstract

Intelligent Domotic Environments are a promising approach, based on semantic models and commercially off the shelf domotic technologies, to realize new buildings that provide users with increased care, adaptability and safety. Intelligent buildings exhibit complex behaviors and are composed of various subsystems, and of many layers and components: such complexity requires innovative design methodologies and tools for ensuring correctness and effectiveness. Therefore suitable simulation and emulation approaches must be adopted, and tools must be provided, to allow architects and designers to experiment with their ideas and to incrementally verify the effects of designed policies. Incremental design and implementation of new strategies or new functionalities requires a validation scenario where the environment is partly emulated and partly composed of real devices. This paper describes a framework, which exploits UML2.0 state diagrams for automatic generation of device simulators from ontology-based descriptions of domotic environments. The device models are managed by the DogSim simulator, that may either simulate a complete building automation system in software, or may be integrated inside the Domotic OSGi Gateway (Dog) as a new emulation driver (EmuDog), allowing partial simulation of virtual devices alongside with actual interface to real devices. The proposed approach has been tested both on a synthetic 6-room flat equipped with 120 devices and on a real home comprising 3 flats located in northern Italy. Synthetic and real experiments show that the approach is feasible and can easily address both simulation and emulation, thus supporting incremental design and validation of home automation policies in realistic home scenarios.

Key words: Simulation, Emulation, Intelligent Domotic Environment, State Chart, Ontology Modeling

1. Introduction

Intelligent Domotic Environments (IDE), i.e., "environments where commercial domotic systems are extended with a low cost device (embedded PC) allowing integration and inter-operation with other appliances, and supporting more sophisticated automation scenarios" [1, 2], currently promise to achieve advanced intelligence at a relatively low cost, enabling the creation of new

*Corresponding author

Email addresses: dario.bonino@polito.it (Dario Bonino), fulvio.corno@polito.it (Fulvio Corno)

building automation scenarios, with much more complex behavior and functionality. The design of such complex systems involves a high number of often conflicting aspects, which include user experience and satisfaction, intelligent behaviors and automated scenarios. Designers should be able to grasp a good understanding of the possible interactions between systems contributing to an IDE design (automatic plants, complex devices, control algorithms, context-dependent scenarios, and the users), and of the modalities with which such interactions will affect the overall correctness and effectiveness of designed solutions. In particular, both modeling tools (for expressing design requirements and programming system behaviors) and validation tools (for simulating different scenarios before actually implementing the system) are a necessary addition to the toolkits of system designers and integrators.

In the literature, very few works address this problem (a short survey is provided in Section 2) and most design methodologies heavily rely on individual designer's experience. To provide a more structured support to the design and test of automation and intelligence for domotic environments, sounder approaches should be developed, based on formal models, property checking, test pattern generation, and simulation based techniques.

Among these approaches, the ability to simulate domotic devices is useful in different design phases (see Table 1): in the preliminary design, a software model of the whole building automation system may be simulated for validating the overall design and the complex interactions that arise when combining advanced devices and control strategies. On the other hand, several scenarios (such as upgrading an existing system or evaluating a new component) require the interaction of real devices with simulated ones: in these cases, simulators must be generated only for the 'missing' devices, while interaction with real devices is provided by the house gateway (emulation).

To clarify the terminology adopted in the paper, we refer to simulation when all represented devices are "virtual" and their execution is done in a completely controlled context, i.e., inside simulation software, such as DogSim [3]. Emulation, sometimes called "Hardware In the Loop simulation," on the other hand, involves at least one real device, i.e., the Home Gateway, and possibly one or more controlled devices.

Table 1: IDE simulation and emulation: applications and required tools.

	Real devices	*Simulated devices*	*Gateway*	*Applications*	*Required tools*
Simulation	No	All	No	Initial abstract design validation	DogSim [3]
Full Emulation	No	All	Real	Validation of interactions with the domotic gateway and its embedded strategies	EmuDog
Mixed Emulation	Some	Some	Real	Validation of incremental deployment and plant evolutions	EmuDog
Deployment	All	No	Real	Actual home control	Dog [2]

This paper proposes a simulation and emulation framework for intelligent environments that may be exploited over the entire IDE life-cycle, from early policy design to final on-the-field deployment. Our approach is based on the automatic generation of state charts from an ontology model (DogOnt [1]) of domotic devices, appliances, and their interconnections. State charts simulate the behavior of each device, as well as the messages and data that they exchange in their in-building installation.

The solutions presented in this paper are incrementally built on technologies developed in the last years: some initial results about state charts simulation were already introduced in a former work by Bonino and Corno [3], the control of real devices is achieved through the open-source Dog gateway [2], and the description of domotic system configurations exploits the DogOnt ontology formalism [1].

In this work, the declarative knowledge encoded in DogOnt is automatically associated to the operational knowledge encoded in device-specific state diagrams by means of ontology querying and customization of diagram templates. Device interconnections, modeled in DogOnt as semantic relations, are translated into event-remapping machines exploiting the event messaging framework defined by the state diagram formalism. Simulation of device behavior is supported at runtime through the DogSim API [3], which embeds a state machine execution engine based on the Apache Commons SCXML library, and by providing event-driven interaction with the IDE model. Emulation capabilities are addressed by means of an emulation driver (EmuDog), integrated inside the Dog [2] Domotic OSGi Gateway.

Experiments involving both DogSim and the EmuDog driver are reported, applying the proposed framework to a synthetic DogOnt model representing a 6-room flat equipped with 120 controllable devices and to 3 domotic flats (Maison Equipée) located in St. Marcel, Valle d'Aosta, Italy. Results show that the approach is feasible and that quite complex behaviors and interactions can easily be simulated, with a satisfying time performance.

The remainder of the paper is organized as follows: Section 2 describes relevant related works. Section 3 briefly provides IDE, Dog, DogOnt and state diagram background. Section 4 introduces state diagrams for Intelligent Domotic Environments, highlighting the underlying assumptions and design choices, while Section 5 shows how state diagram descriptions of IDEs are automatically generated from DogOnt instances. Section 6 describes the architecture of the DogSim API whereas Section 7 describes the EmuDog driver design and its integration in Dog. Section 8 provides experimental results. Eventually, Section 9 draws conclusions and proposes future works.

2. Related Works

Despite the increasing interest in the AmI field and on related technologies (see [4] for a recent review), the current state of the art in modeling and simulation for AmI environments, and in particular for IDEs, is rather sparse and offers a mixture of approaches from many different communities whilst a general consensus is still lacking. Emulation is even more neglected and nearly no approaches have been currently presented that address this specific aspect of intelligent environment design.

Among the proposed approaches, the Habitation language [5] is one of the most affine works to DogSim. Habitation is a Domain-Specific Language (DSL) specifically designed for Home Automation. It is one of the earliest attempts to apply Model Driven Engineering to the Home Automation domain. Habitation aims at tackling the life-cycle of home automation system design. It combines a model-driven approach with DSLs to support the design of home automation systems, from high-level, technology independent graphical design to technology-specific automatic code generation. The Habitation language is based on a three-layered approach including: 1) a computation-independent model, which represents the syntax and part of the semantics of the defined DSL, 2) a platform-independent model, which is a simplification of the UML metamodel for reactive systems [6], and considers components, activities and state diagrams, and 3) a

platform-specific model in which a meta-model for KNX/EIB[1] home automation technology is defined, exploiting the domain object model used by ETS[2]. The approach proposed in this paper has several differences and complementary aspects. First, Habitation aims at supporting the Home Automation design from requirements specification to programming of individual devices but completely ignores validation and simulation of the generated system configurations. On the contrary, the proposed approach is focused on the last two aspects thus being complementary with Habitation. Secondly, Habitation bases its abstraction model on UML, while DogSim exploits OWL descriptions of the home environment. UML is more suited for code generation but it does not support the design and implementation of complex inferences and artificial intelligence policies, on the other hand OWL can be exploited both to design inferences and to define meta-models for code generation. Finally, the Habitation approach is strongly targeted at generating software controllers for single devices; however, IDEs are often made by not-programmable devices whose predefined behaviors have to be combined in new ways, for achieving more "intelligent" functionalities.

In [7], Conte et al. tackle the problem of modeling and simulating Home Automation Systems (HAS) by applying the Multiple Agent Systems (MAS) theory [8]. They start from a formal theory of Home Automation Systems [9], based on the MAS theory and construct an environment for simulation and emulation of HAS. On top of this theory, Conte et al. define performance indicators for paradigmatic control strategies allowing to simulate and analyze the effects caused by control parameters variations on the control overall performance. The home simulation environment they introduce is a specialization of general MAS simulation engine in which agents represent appliances, described from an abstract, behavioral point of view. Each agent's behavior is modeled as a sequence of transitions from state to state driven by a set of rules requiring the availability of specific resources. The sequence of state transitions during normal operation of each agent is time-driven, except when required resources are, or become, unavailable. The main focus is on simulating single home control tasks as a set of independent, time-driven processes concurrently accessing and consuming a shared set of resources, e.g., electricity, hot water, etc. Although both Conte's approach and DogSim are based on state machines, they show many differences. First, while the goal of the Conte's simulation environment is the refinement of control strategies, DogSim goal is to provide a simulation environment for real-world IDEs where devices are not isolated but can interact explicitly through commands and notifications. Second, DogSim is designed to provide a validation and simulation framework on top of which specific aspects of IDE control and intelligence can be investigated; in this sense the approach of DogSim is compatible and complementary with Conte's work. Finally, in the approach presented in [9] the formal model of the HAS does not take into account the environment configuration nor the functionalities of real-world domotic plants, which in most cases are much simpler than the domotic agents they define. Moreover the MAS model they define does not provide easy to exploit mechanisms for reasoning on the environment configuration and state, thus limiting the kind of applicable intelligent behaviors.

3. Background

Domotic systems, also known as home automation systems, have been available on the market for several years, however only in the last few years they started to spread over residential

[1] KNX/EIB, http://www.eiba.com/en/eiba/overview.html
[2] The official configuration software for KNX/EIB plants.

buildings, thanks to the increasing availability of low cost devices and driven by new emerging needs on house comfort, energy saving, security, communication and multimedia services.

Current domotic solutions are still suffering from two main draw-backs: they are produced and distributed by various electric component manufacturers, each having different functional goals and marketing policies; and they are mainly designed as an evolution of traditional electric components (such as switches and relays), thus being unable to natively provide intelligence beyond simple automation scenarios. The first drawback causes interoperation problems that prevent different domotic plants or components to interact with each other, unless specific gateways or adapters are used. While this was acceptable in the first evolution phase, where installations were few and isolated, now it becomes a very strong issue as many large buildings are mixing different domotic components, possibly realized with different technologies, and need to coordinate them as a single system. On the other hand, the roots of domotic systems in simple electric automation prevent satisfying the current requirements of home inhabitants, who are becoming more and more accustomed to technology and require more complex interaction possibilities.

To overcome this issues while retaining low cost and wide availability typical of commercially off-the-shelf home automation, in 2008 we defined the concept of Intelligent Domotic Environment [2]. An Intelligent Domotic Environment is a home or building where existing domotic systems (wired or wireless), are extended by adding devices and agents that support interoperation and intelligence. The core of an IDE is a machine processable formal model of the environment expressed in form of ontology. Ontologies allow on one hand to formally represent the environment components, i.e., devices, appliances, furniture elements, in a technology independent and easy to elaborate format. On the other hand, they provide support for advanced intelligence based on logic and rule-based reasoning, context representation, etc.

Rather than accessing and handling devices on the basis of specific protocols and communication technologies, an IDE abstracts the low-level communication issues typical of such complex environments and offers a uniform and abstract view of the home (or building) and of the devices therein. Any device therefore becomes accessible on the basis of its capabilities, independently from the specific technology with which the device is built. For example, in an IDE, a lamp will always be represented as an object that can either be switched on or off, and that if switched on emits light.

To support this abstraction chain, three main components are needed (Figure 1): a set of network-level drivers able to deal with all low level issues that the IDE shall overcome, a domain model specifying the capabilities of all the entities taking part to the IDE, and a Home Gateway encapsulating both the drivers and the model, thus offering flexible access to the IDE functionalities. The latter component is the natural place where intelligent policies and behaviors can be deployed.

While IDEs easily solve interoperation issues and offer clear ways for designing and implementing home intelligence on top of commercial domotic systems, there is still a lack of tools to support end to end development of these environments, from early design to final on-the-field installation. Currently, a IDE designer can already exploit DogOnt [1] as a rather stable environment modeling ontology and Dog [2] as beta-level open source gateway implementing the above principles. Home descriptions shall be defined during the design process, however the designer is not required to master ontologies and related formalisms since suitable tools[3] (auto-generation rules, XML-to-OWL translators, etc.) are available for reducing the complex ontology instantiation process to a simpler definition of an XML list of devices. We are currently working on

[3] included in the Dog gateway distribution at `http://domoticdog.sourceforge.net`

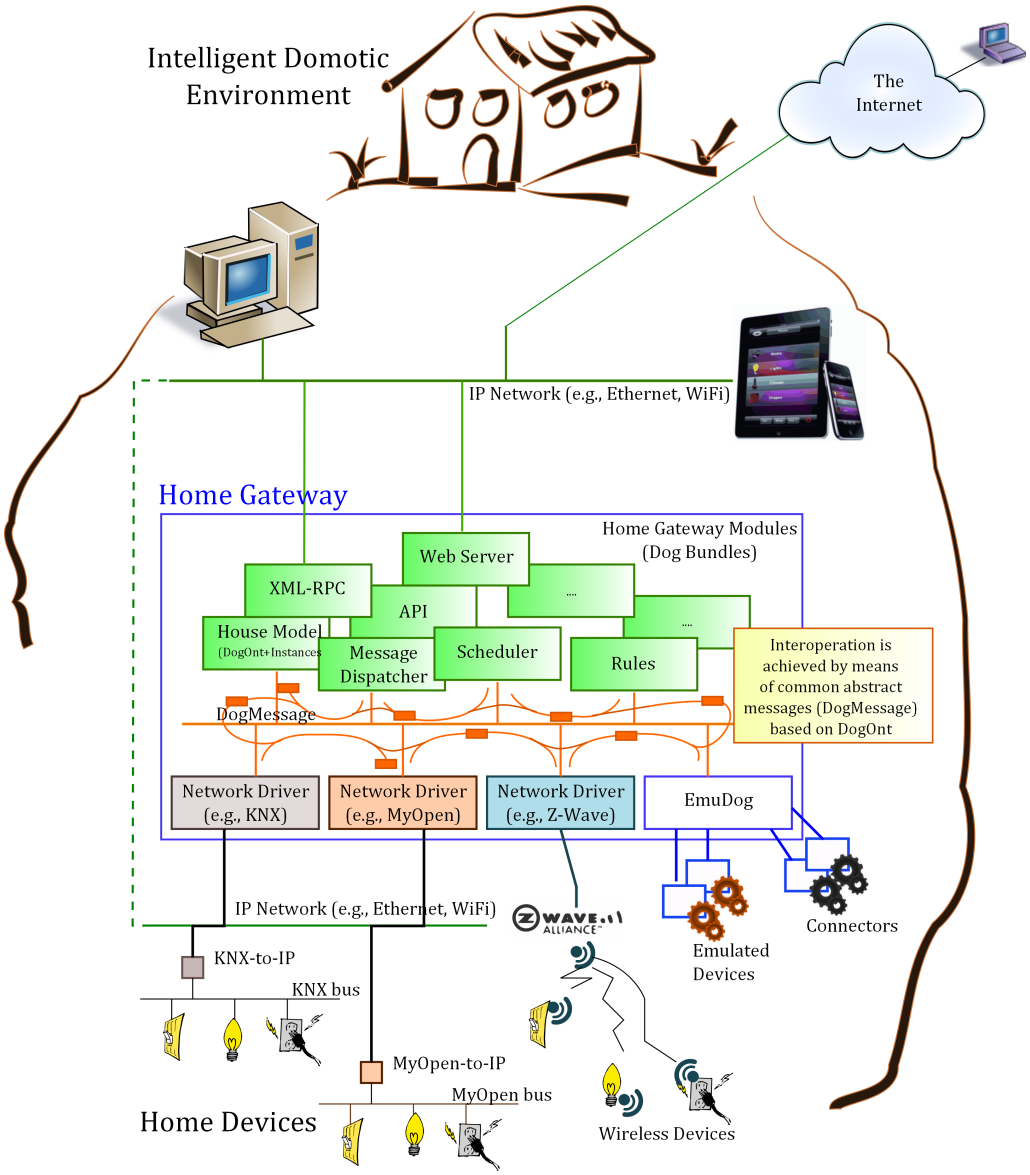


Figure 1: The general IDE architecture.

graphical modeling environments for enabling architects and interior designers to easily master next generation intelligent domotics.

Besides modeling, there is a growing need to simulate the environment for which intelligent policies are designed. Such a simulation is almost complete in the early design stages while (see Table1), during IDE deployment, intelligent behaviors need to be refined by interfacing real devices (hardware in line simulation). Next sections show how this can be achieved by exploiting the DogOnt environment model and the Harel's [10] state diagram formalism. For the benefit of readers not familiar with these technologies, the rest of this section provides a short introduction to DogOnt-based IDE modeling, Dog and Harel's state diagrams, with a particular reference to the specific semantics defined in the UML 2.0 standard [11].

3.1. DogOnt in a nutshell

DogOnt is an ontology specifically designed for modeling Intelligent Domotic Environments (for a complete description of the DogOnt design and modeling capabilities see [1]). It is organized along 5 main hierarchies of concepts (Figure 2) supporting the description of:

- the domotic environment structure (rooms, walls, doors, etc.), by means of concepts descending from *BuildingEnvironment*;
- the type of domotic devices and of smart appliances (concepts descending from the *Controllable* subclass of the *BuildingThing* main concept);
- the working configurations that devices can assume, modeled by *State*s and *StateValue*s (see the following paragraphs for more details);
- the device capabilities (*Functionalities*) in terms of accepted events and generated messages, i.e., *Command*s and *Notification*s;
- the technology-specific information needed for interfacing real-world devices (*NetworkComponent*) and
- the kind of furniture placed in the home (concepts descending from the *UnControllable* subclass of the *BuildingThing* main concept).

DogOnt models domotic devices in terms of functionalities and states.

Functionalities. They 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.

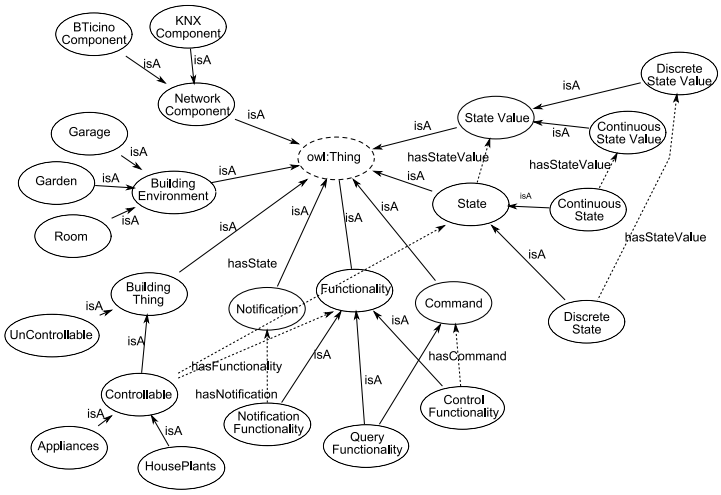


Figure 2: DogOnt in a nutshell.

Functionalities are either associated with commands (for *ControlFunctionalities*) or with notifications (*NotificationFunctionalities*) that further detail the specific operations supported by DogOnt device instances. Figure 3 shows a sample DogOnt model of a dimmer lamp, with functionalities highlighted in bold.

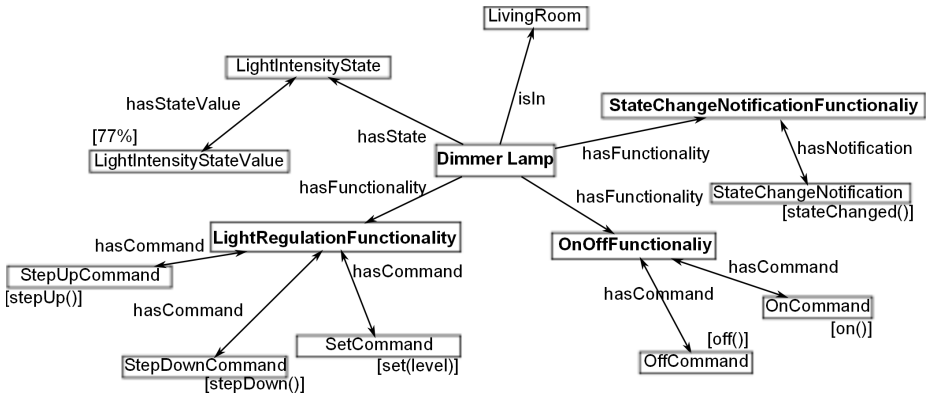


Figure 3: A sample Dimmer Lamp model in DogOnt

Device interconnections are modeled by the *controlledObject* relationship [12] linking a controller[4] device (e.g., a switch) to one or more controlled devices[5](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. Connections can be further specialized through the *generatesCommand* relation, which permits to specify the command(s) generated in response to a given device

[4] `rdfs:domain(dogont:Control)`
[5] `rdfs:domain(dogont:Controllable)`

notification (Figure 4).

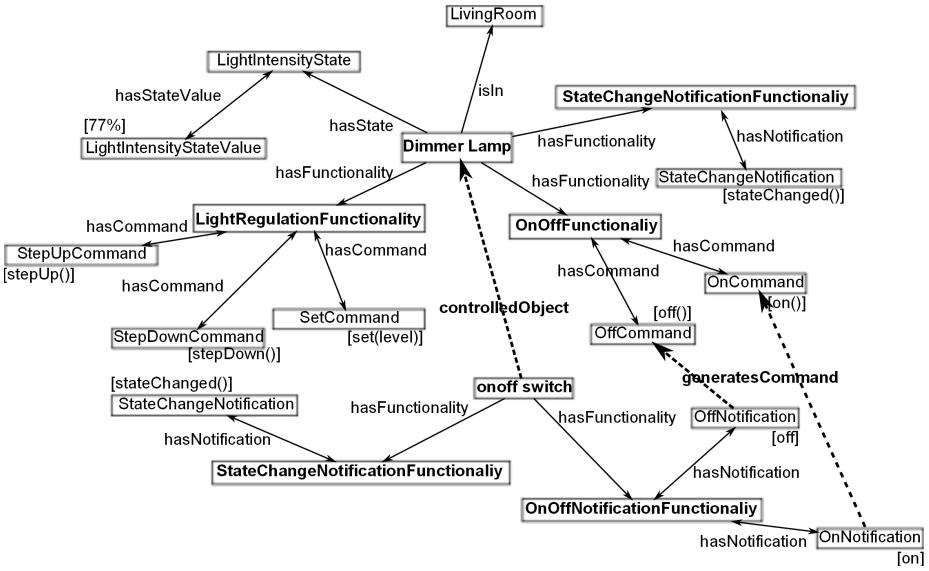


Figure 4: A sample of connection modeling in DogOnt where a Switch controls a Dimmer Lamp.

States. They 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: being on or off, the number of the CD track being played, and having a given audio volume are all independent stable configurations of a CD player. In DogOnt such behaviors are called `dogont:State`. The description of each `dogont:State` is represented by a set of identifiers, called `dogont:StateValue`, that model each operating condition. For example the CD player is modeled as having three independent `dogont:States`: `dogont:OnOffState`, `dogont:PlayingState` and `dogont:VolumeLevelState`. Each of these three states include a specific set of possible state values (for example, the first state includes a `dogont:OnStateValue` and a `dogont:OffStateValue`). The current state of a device is therefore defined by a list containing one `dogont:StateValue` per each `dogont:State`.

Example 1. Consider a shutter actuator model as an example (Figure 5). The shutter actuator is represented in DogOnt as having one `dogont:UpDownRestState` that, in turn is related to 3 state values: `UpStateValue`, `DownStateValue` and `RestStateValue`. These values represent the visible operating conditions of the actuator. However, the real device can have more behaviors than the model: for example it can operate in 2 more conditions identifying if the actuator is raising or lowering the shutter, respectively. These hidden conditions are not modeled in DogOnt and therefore they have no counterpart in the *StateValue*s associated to the `dogont:UpDownRestState` concept.

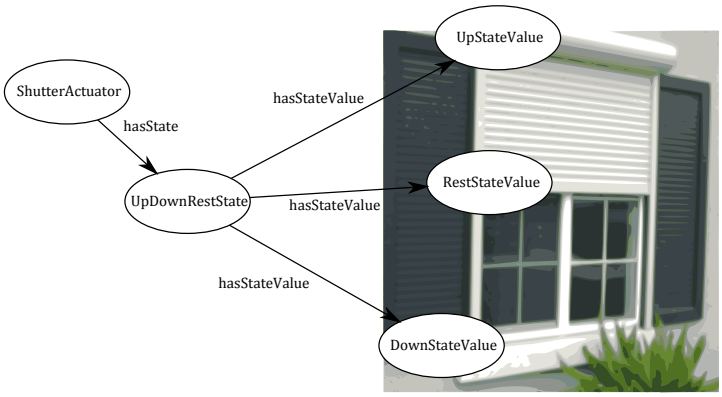


Figure 5: State modeling of a roller shutter actuator.

3.2. Dog

Dog (Domotic OSGi Gateway) [2] is a Home Gateway [13, 14] designed to transform new, or existing, domotic installations into intelligent domotic environments. The main features of Dog include: interoperability, versatility, advanced intelligence and accessibility, in terms of supported applications and affordable costs. Interoperability and versatility are achieved by means of a strongly modular architecture exploiting the OSGi framework, the de-facto reference framework for Residential Gateways [15]. Advanced intelligence support is gained by formally modeling the home environment through the DogOnt ontology and by defining suitable reasoning mechanisms on top of it. Accessibility is provided through an application-level API exposed either directly, to external OSGi bundles, or wrapped into an XML-RPC [16] service access point. Costs are kept low by ensuring the ability to run the Dog gateway on cheap hardware platforms, e.g., netbooks[6] or embedded computers.

The Dog logic architecture is deployed along 4 layers (rings, in the Dog terminology, see Figure 6) ranging from low-level interconnection (Rings 0 and 1) to high-level modeling and interfacing (Rings 2 and 3). Each ring hosts several OSGi bundles implementing the platform modules.

Specifically, Ring 0 includes the Dog common library and the bundles required for managing the interactions between the OSGi platform and the other Dog bundles. Ring 1 encompasses the Dog bundles that provide interconnection services for domotic technologies.[7] Each network technology is managed by a dedicated driver, which abstracts the network protocol into a common, high-level message protocol (DogMessage, see Figure 1) based on the DogOnt ontology model. Ring 2 provides the routing infrastructure for messages traveling across network driver bundles and directed to (or originated from) Dog bundles. It hosts the Dog core intelligence, based on the DogOnt ontology and implemented by the House Model bundle, and the Dog runtime rules core: DogRules. Finally, Ring 3 hosts the Dog bundles offering access to external applications, either by means of the API bundle, for OSGi applications, or by exploiting an XML-RPC service access point for applications based on other technologies.

[6]The reference platform is currently an ASUS eeePC 701 netbook, with 4GBytes of SSD and 512 Mbytes of RAM

[7]Currently Dog supports KNX and MyOpen domotic networks, and the Z-Wave wireless protocol; ZigBee support is under development

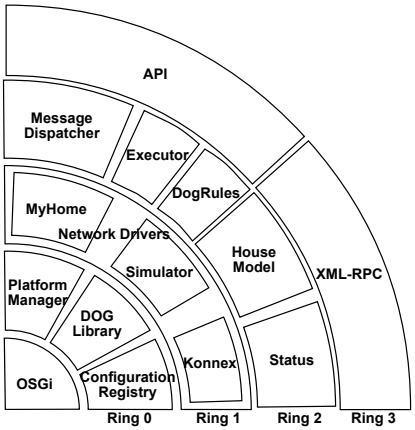


Figure 6: The Dog gateway architecture.

3.3. State diagrams

UML state diagrams [11, 17, 18] are based on the state chart notation [10] invented by David Harel in 1987. State diagrams are designed to specifically address the problem of representing large and complex reactive systems, avoiding state explosion issues by means of a suitable concurrent semantics. According to Harel "A reactive system is characterized by being to a large extent event-driven, continuously having to react to external and internal stimuli" [10]. Intelligent Domotic Environments can be considered as belonging to the class of reactive systems, at least for what concerns their control part. They must in fact respond to user actions, to environmental changes (external stimuli) and to events generated by control algorithms (internal stimuli). Although some approaches exploit state diagrams for automatically generating home automation circuits and software (e.g., [19]) we have no knowledge of other approaches applying them to formal IDE modeling.

4. State Diagrams for IDE

Design of Intelligent Domotic Environments mainly involves 3 different areas: control, data acquisition and media handling. In this paper we concentrate on issues related to the control domain, and we apply and extend the DogSim state diagram model for interconnected IDE devices, first introduced in [3].

4.1. Device modeling

In DogSim, every DogOnt device concept has a state machine counterpart modeling the dynamic behavior (in terms of states transitions) of devices belonging to that class. For example, the `dogont:Lamp` class has a matching state machine that models the dynamic behavior of a generic lamp (Figure 7).

While the state machine topology is directly related to the device class and to the functionalities and states modeled in DogOnt, every single device instance deployed in a specific IDE possesses different states and can transition from one state to another independently from the other devices of the same class. In other words, while DogOnt device concepts define the general

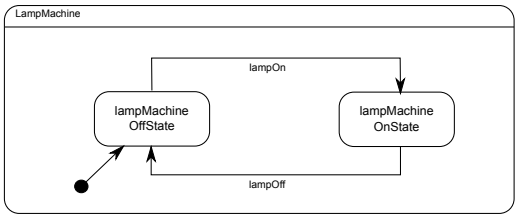


Figure 7: The state machine template associated to the `dogont:Lamp` class

state machine topology, called in this paper "state machine template", DogOnt device instances correspond to distinct state machines, instantiating the former template.

Templates are defined during the DogOnt design process and come bundled with the environment description primitives that the ontology provides. They define the skeleton of a state machine representing every device behavior and include suitable customization points to support the machine instantiation in specific models. Figure 8 shows a sample `WindowActuator` template, encoded in SCXML. Customization points are represented by means of a placeholder XML entity (`&id;`).

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE scxml SYSTEM "template.dtd">
<!-- @device=WindowActuator -->
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0">
 <state id="&id;windowMachine">
  <state id="&id;openState">
   <transition event="&id;close" target="&id;closingState"/>
  </state>
  <state id="&id;closeState">
   <transition event="&id;open" target="&id;openingState"/>
  </state>
  <state id="&id;closingState">
   <transition event="&id;windowClose" target="&id;closeState"/>
   <transition event="&id;tClose" target="&id;closeState"/>
   <transition event="&id;open" target="&id;openingState"/>
   <onentry>
    <send sendid="&id;delay1" targettype="scxml" event="&id;tClose" delay="2000"/>
   </onentry>
   <onexit><cancel sendid="&id;delay1"/></onexit>
  </state>
  <state id="&id;openingState">
   <transition event="&id;windowOpen" target="&id;openState"/>
   <transition event="&id;tOpen" target="&id;openState"/>
   <transition event="&id;close" target="&id;closingState"/>
   <onentry>
    <send sendid="&id;delay2" targettype="scxml" event="&id;tOpen" delay="2000"/>
   </onentry>
   <onexit><cancel sendid="&id;delay2"/></onexit>
   </state>
   <initial><transition target="&id;closeState"/>
         </initial>
 </state>
</scxml>
```

Figure 8: Sample WindowActuator template (graph and SCXML).

During the design of a specific IDE, ontology classes are instantiated and generic templates (as the one shown in Figure 7) are referred to specific instances (e.g., to the "MyDesktopLamp") by

simply renaming every state and every transition accordingly (Figure 9), i.e., by substituting the "`&id;`" placeholder with the device instance unique name (URI).

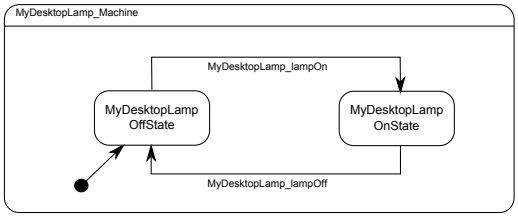


Figure 9: The state machine associated to the `myhome:MyDesktopLamp` instance of the `dogont:Lamp` device class.

Whenever a new device class is added to the ontology, the corresponding state machine template is developed by the ontology design team thus enabling IDE designers to concentrate on specific IDE descriptions, without requiring intense and technically difficult modeling efforts. Currently 102 template machines are defined, one per each device class descending from `dogont:Controllable`. The joint availability of DogOnt device classes and corresponding state machine templates supports the development of graphical design environments for architects and interior designers, which are not required to master any ontology related knowledge.

Different, possibly interacting, device instances are modeled as different state machine instances combined through parallel composition, i.e., by representing them as orthogonal substates of the full environment machine (see Figure 10).

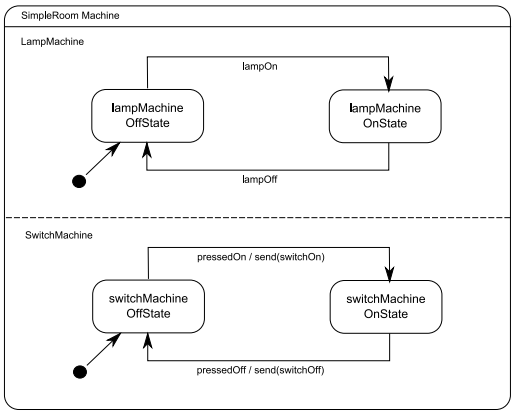


Figure 10: A minimal IDE state diagram.

IDE state machines can receive and send events from/to both the surrounding world and the orthogonal sub-states (regions in the UML notation) composing them. In the simple room example, reported in Figure 10, the whole IDE machine accepts 4 input events (`pressedOn`, `pressedOff`, `lampOn` and `lampOff`) and may generate 2 output events represented by the arguments of the send construct, i.e., `switchOn` and `switchOff`. The `pressedOn` and `pressedOff` events are external and user-generated, while the others are internal events abstracting the actual device operations. According to the UML model semantics, connections between different state machines are represented by identical naming of corresponding input and

output events.

Example 2. Consider the example in Figure 4 where a simple switch and a dimmer lamp are modeled in DogOnt as two connected devices. Figure 11 shows a possible state machine repre-

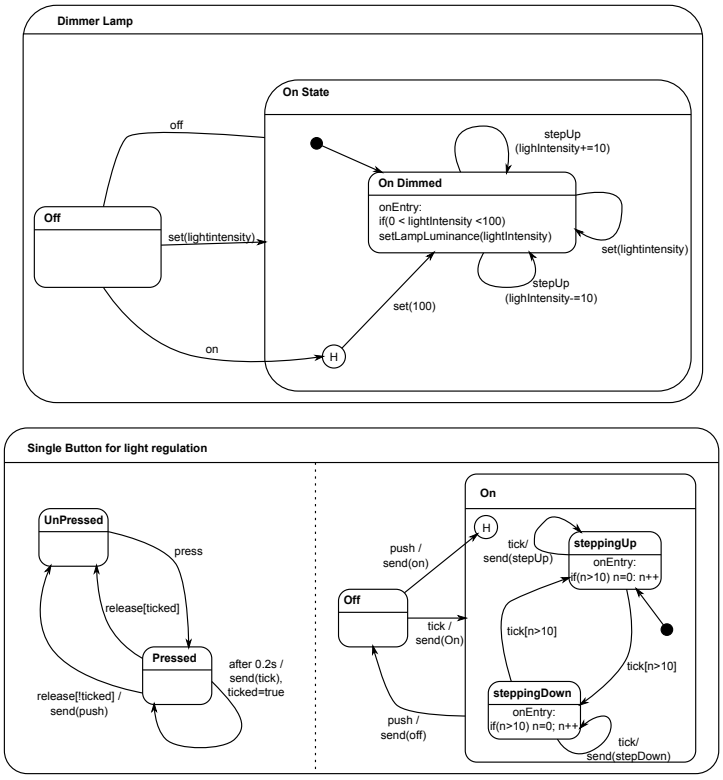


Figure 11: Example of non-trivial home device statemachines.

sentation of the devices used in that model, using non-trivial state machine templates.

In the top of Figure 11 the dimmer lamp state machine is modeled: it possesses one off and one on state (`dogont:OnOffState`) and the on state has one "On Dimmed" sub state modeling the current intensity of the lamp (see the state variable named "lightintensity"). This inner state corresponds to the `dogont:LightRegulationState` concept defined in DogOnt. Moreover the dimmer lamp possess a memory able to store the light intensity value when the lamp is switched off. Such a property is typical of real-world dimmer actuators and it is easily implemented in the state chart model reported in Figure 11 by exploiting a history node. Whenever the history is empty, i.e., the dimmer lamp ha never been regulated to a dim value, the transition between the history node and the OnDimmed state causes the lamp to be switched on at the maximum intensity.

The bottom of Figure 11 shows an even more complex state machine template modeling the typical operation of a single-button dimmer switch. A single button dimmer switch works as follows: one quick pression of the button allows to switch the lamp on and off, while keeping

the button pressed allows to regulate the light intensity through a ramping mechanism that cyclically increases and decreases the lamp illumination in a triangular-shaped intensity modulation. The state machine reported in Figure 11 simulates the real switch behavior by exploiting a parallel state machine composed of two parts tackling the button pressure and the ramp generation, respectively. Whenever the button is pressed ("press" event in the machine) the button modeling machine (on the left) starts a cycling polling (modeled using the "after" event defined in UML2.0 state diagrams) to check if the button is still pressed. While the button is pressed, the polling cycle generates a tick event which is received by the right part of the state machine, which is responsible for the ramp generation. The intensity ramp is obtained by "oscillating" between a "Stepping Up" state, sending 10 tick-synchronized "stepUp"commands to the connected dimmer lamp, and a "SteppingDown"state sending 10 "stepDown" commands. The ramp is repeated until the button is pressed, i.e., until the tick event is received. Finally, when the button is released, a push event is sent only if no tick occurred, thus accounting for the behavior associated to short button clicks.

4.2. Connection modeling

Real world device-to-device communication is complex and cannot be completely addressed through standard identical naming defined in UML, e.g., the general case of n-to-n connections cannot be addressed. Therefore, in DogSim we model device interconnections by designing new, special purpose, orthogonal machines (connectors) acting as event-translators, i.e., converting each input event into a corresponding set of output events.

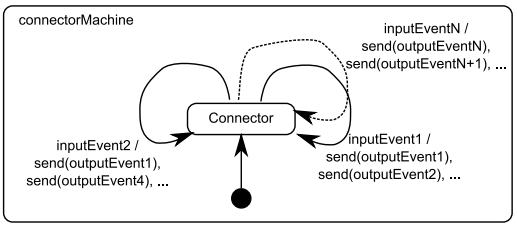


Figure 12: The connector machine skeleton.

The anatomy of a connector machine is extremely simple: it includes a single state and a variable set of self-loop transitions translating a given input event into corresponding output events (Figure 12). Connectors support modeling of any kind of state machine interactions with a negligible complexity increase in the resulting IDE state diagram.

Example 3. Referring back to DogOnt, while device machines correspond to DogOnt device classes, connector machines are used to model, in state diagrams, the same information represented by the union of the `dogont:controlledObject` and the `dogont:generatesCommand` relationships. The former identifies which devices shall be interconnected, i.e., the state machine names (used in all machine events, as better explained in the following subsection). The latter specifies which notifications and commands shall be connected, i.e., which events of the two machines shall be bridged. For example, the connection between the switch and the dimmer lamp shown in Figure 4 is represented by in the connector machine shown in Figure 13 where the OnNotification of the switch has been mapped onto the `switch_onEvent` and the OnCommand of the dimmer lamp has been translated to the `dimmerLamp_onEvent`. The same procedure

allows tackling the connection between the OffNotification of the switch and the OffCommand of the lamp.

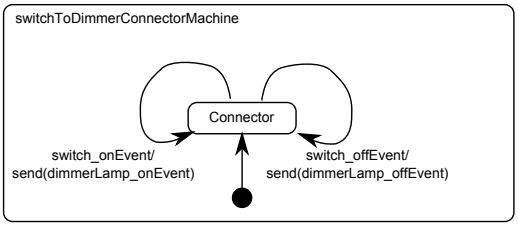


Figure 13: The connector machine for the sample switch and dimmer lamp state machines.

It must be noticed that connector machines act on the abstract network representation deriving from DogOnt. As a consequence their behavior is completely independent from any network specific issues which are demanded to the abstraction layer that translates DogOnt-based commands and notifications to specific home-automation protocols. In case the IDE is powered by a Dog instance, this abstraction layer is composed by all the available network drivers, while the connector machines work only on technology neutral DogMessages.

5. Automatic Generation of IDE State Diagrams

State machines and connectors can be automatically generated given a complete IDE description in DogOnt, i.e., given the DogOnt ontology schema and instances. The DogOnt schema defines the set of device templates, associated to every DogOnt class during the ontology development process. The specific device instances, instead, are extracted by querying the IDE model and their peculiar characteristics are exploited to specialize general templates into specific machine instances. Connections between devices are extracted in the same way (ontology querying) and automatically converted into connector instances. Discussing the automatic generation process is out of scope in this paper (more details can be found in [3]), however a brief summary of the generation workflow is reported.

Figure 14 shows the flow chart of the automatic generation algorithm. It works as follows: first the ontology information (concepts and instances) is checked for consistency, reasoned to extract implicit knowledge and transitively closed over the `isA` (hierarchy) relationship. Then, the ontology is queried using a suitable set of SPARQL queries as part of the DogSim simulator. (Figures 15,16) extracting the list of all `dogont:Controllable` instances, i.e., the list of all devices that the IDE home gateway can potentially control. Subsequently, the automatic generation process extracts from the DogOnt template library the state machine template associated to each of the found device instances. If the IDE ontology model only uses device classes explicitly defined in DogOnt a corresponding template is always found. On the converse, if a given device is defined as belonging to a user-defined subclass of a DogOnt device concept, e.g., as a user-defined`ColoredFlashLamp` subclass of the standard `dogont:FlashingLamp`, no template can directly be found. In such a case, an automatic generalization process starts and navigates the device class hierarchy until a state machine template is available. In the worst case, no template is found until the root of device concepts (`dogont:Controllable`) is reached; if this happens the device state machine is not created and the generation process continues on the next device instance. After the device machine generation, the ontology is queried again

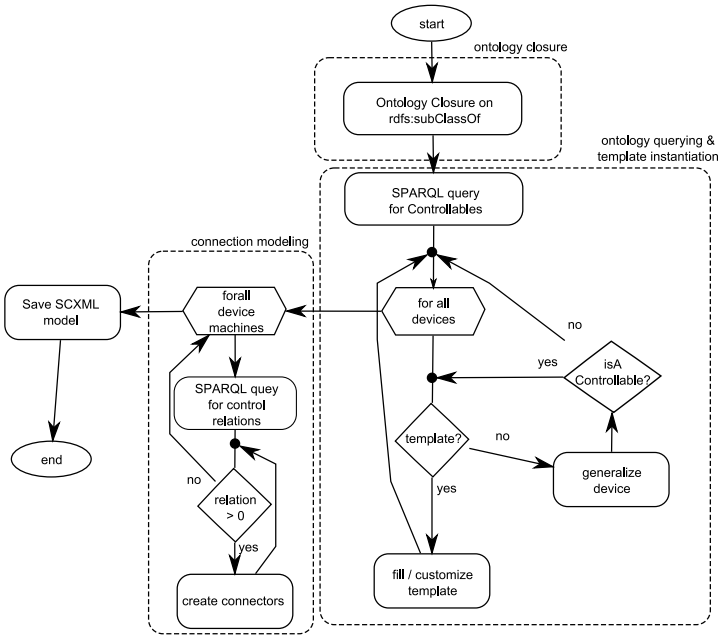


Figure 14: The automatic state diagram generation process.

(Figure 16) to extract device interconnections, i.e., to find device instances connected by means of the `dogont:controlledObject` and the `dogont:genereatesCommand` relations. For each interconnection, suitable connectors are generated adding device-to-device interactions to the generated IDE simulator. In the end, the complete IDE state diagram is saved as a standard SCXML machine definition [20] and shared with other modules or applications.

```
SELECT ?controllable WHERE
{
?controllable rdfs:subClassOf dogont:Controllable
}
```

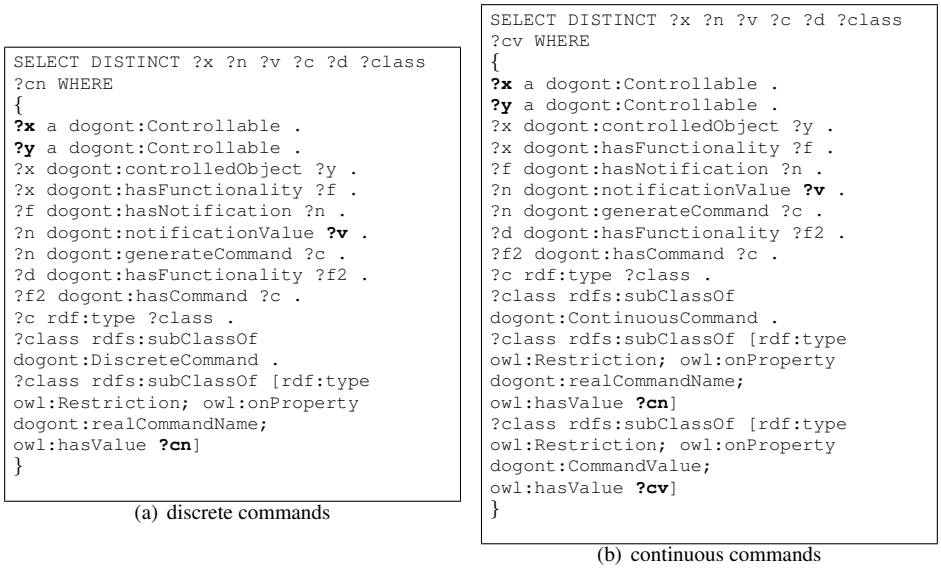
Figure 15: A Sample SPARQL query for extracting all controllable device classes.

6. Simulation

Automatic generation of IDE state diagrams and state diagram execution (simulation) is implemented by the DogSim Simulation API. This section, in particular, analyzes the *Simulation* case (Table 1), where all the IDE components are simulated and no real devices nor gateways are deployed. Section 7 will later show how DogSim has been exploited to implement emulation inside the Dog gateway.

DogSim is logically organized into 3 layers (Figure 17):

- the *data layer* hosting the DogOnt instance to be simulated, the repository of device templates and the resulting IDE SCXML machine;



```
SELECT DISTINCT ?x ?n ?v ?c ?d ?class
?cn WHERE
{
?x a dogont:Controllable .
?y a dogont:Controllable .
?x dogont:controlledObject ?y .
?x dogont:hasFunctionality ?f .
?f dogont:hasNotification ?n .
?n dogont:notificationValue ?v .
?n dogont:generateCommand ?c .
?d dogont:hasFunctionality ?f2 .
?f2 dogont:hasCommand ?c .
?c rdf:type ?class .
?class rdfs:subClassOf
dogont:DiscreteCommand .
?class rdfs:subClassOf [rdf:type
owl:Restriction; owl:onProperty
dogont:realCommandName;
owl:hasValue ?cn]
}
```

(a) discrete commands

```
SELECT DISTINCT ?x ?n ?v ?c ?d ?class
?cv WHERE
{
?x a dogont:Controllable .
?y a dogont:Controllable .
?x dogont:controlledObject ?y .
?x dogont:hasFunctionality ?f .
?f dogont:hasNotification ?n .
?n dogont:notificationValue ?v .
?n dogont:generateCommand ?c .
?d dogont:hasFunctionality ?f2 .
?f2 dogont:hasCommand ?c .
?c rdf:type ?class .
?class rdfs:subClassOf
dogont:ContinuousCommand .
?class rdfs:subClassOf [rdf:type
owl:Restriction; owl:onProperty
dogont:realCommandName;
owl:hasValue ?cn]
?class rdfs:subClassOf [rdf:type
owl:Restriction; owl:onProperty
dogont:CommandValue;
owl:hasValue ?cv]
}
```

(b) continuous commands

Figure 16: Sample SPARQL query for extracting connector data.

- the *model generation layer* hosting the software modules for template retrieval and customization (Template Factory) and for implementing DogOnt querying and reasoning functionalities (DogOnt2SCXML);
- the *simulation layer* where the state diagram execution engine is located.

External applications can access services offered by DogSim through an event-based service access point (defined as a set of Java interfaces) shown on top of the 3rd layer in Figure 17.

6.1. Data Layer

The data layer has a twofold nature: it contains all the information sources needed for generating the state diagram model of a given IDE (*generation*) and it provides state diagram persistence on file (*simulation*). It wraps 3 different resources:

- the *DogOnt* structure and instantiation modeling the given IDE;
- the *template repository*, currently implemented as a simple directory tree containing template SCXML files;
- the IDE SCXML resulting from the generation step.

While the DogOnt instance and the Template Repository are mandatory and must be present to enable a successful model generation, the final SCXML state diagram may either be kept in memory for on the fly simulation or can be persisted on a SCXML file for enabling further reuse.

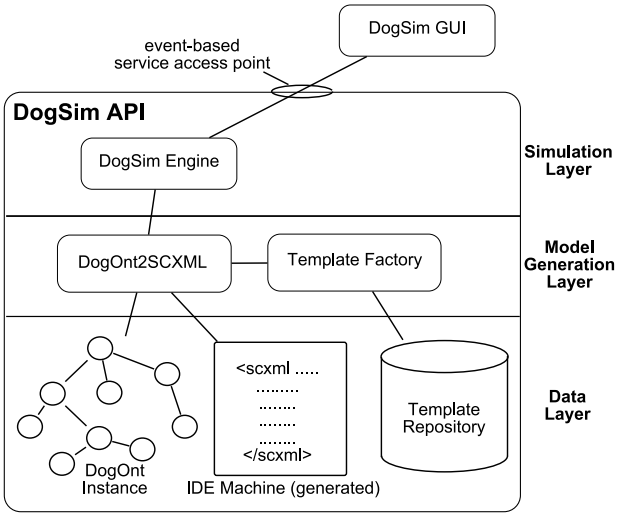


Figure 17: The DogSim logic architecture.

6.2. Model Generation Layer

The model generation layer hosts the two modules of DogSim: *TemplateFactory* and *DogOnt2SCXML* and they implement solutions described in Section 4. The *TemplateFactory* manages the Template Repository data and facilitates template retrieval based on the device instances; it only supports direct template to device class mappings. Whenever a template is requested, the Template Factory queries the Template Repository and if a match is found, returns a customized template representing the device instance provided as input. If no matching template is found, the module returns a retrieval error which can be managed by the calling software module, e.g., for triggering a generalization step, as done in *DogOnt2SCXML*.

DogOnt2SCXML manages ontology-related information by exploiting the HP Jena API [21] to load and represent a DogOnt instantiation as a set of Java objects. Jena supports the execution of the transitive closure of the DogOnt model by either exploiting an internal or an external reasoner.[8] In addition, it facilitates queries (SPARQL) based on the reasoned model. DogOnt2SCXML implements the automatic generation algorithm described in Section 4, including the automatic generalization of devices for which specific templates are not available. It can either work on-the-fly, i.e., by generating an in-memory representation of the IDE state diagram model, ready to be used for simulation, or it can generate a persistent SCXML [20] representation of the IDE, which can either be fed to the DogSim engine or that can be reused at later times. In both cases, DogOnt2SCXML exploits the services offered by the Template Factory for creating the needed device machines and for instantiating connectors.

6.3. Simulation Layer

The simulation layer provides the functionalities needed for executing runtime simulations of generated IDE state diagrams and offers an event-based interface allowing external applications

[8]See [3] for a comparison of reasoning performance in automatic state diagram generation for Pellet [22] and the internal Jena reasoner [21].

to dynamically interact with the simulation. It exploits the Commons SCXML Apache library[9] which implements an SCXML 1.0 compliant simulation engine and provides hooks for listening to events and state changes of the simulated machines, and for injecting new events. Such an engine is wrapped to provide a convenient, event-based, access API and to provide support functionalities for either loading in-memory state machines generated by the generation layer or for loading SCXML IDE machines stored on files.

7. Emulation

Emulation of IDE devices (rows 2 and 3 in Table 1) is addressed by integrating the DogSim API and the Dog domotic gateway. This integration allows exploiting the simulation abilities of DogSim in a mixed environment where state machine events can both be synthetic (as in DogSim) or can stem from real device interactions thanks to the Dog messaging framework. During emulation, the control and intelligence parts of Dog deal uniformly with real and simulated devices, and which devices are actually simulated is unknown to the upper layers of Dog. In fact, emulated devices are handled by a special "network driver," (in Ring 1) that, from the point of view of the gateway, is indistinguishable from other network drivers used to control real plants. This new network driver, called EmuDog (Figure 18), integrates the DogSim Simulation API in Dog by interfacing with the Dog event messaging bundle, *MessageDispatcher* and, the *DogLibrary* and *HouseModel* bundles for handling ontology and templates.

Figure 18 shows how DogSim modules have been wrapped to interface Dog bundles. This interfacing sugar is needed to let DogSim to either work as a standalone simulation environment or as on-line emulation driver in Dog. While the overall logic architecture is the same in both cases, in EmuDog some ontology-related tasks are delegated, by means of the *HouseModel Adapter*, to the Dog gateway, which already performs ontology closure and supports ontology querying through the *House Model* bundle. Additionally, the HouseModel Adapter enables the EmuDog driver to only attach the subset of devices needing emulation, while real devices are directly managed by the corresponding network drivers (e.g., MyHome or KNX).

The EmuDog life-cycle inside Dog is divided into 2 main phases: initialization and runtime operation.

In the initialization phase (Figure 19), the HouseModel loads the DogOnt ontology and the corresponding instances and performs the transitive closure of the ontology. Then, EmuDog queries the House Model discovering devices needing emulation, i.e., devices for which no network driver has been found or devices that explicitly require emulation through a special property `Dogont:emulateThis` being set.

For each found device, EmuDog builds the corresponding state machine, exploiting the DogOnt2SCXML module.

Connector machines are then generated according to the following filtering rules:

a) If both connected devices are real, no connector is generated.

b) If at least one of the involved devices is virtual (emulated) then a suitable connector machine is created.

[9] `http://commons.apache.org/scxml/index.html`

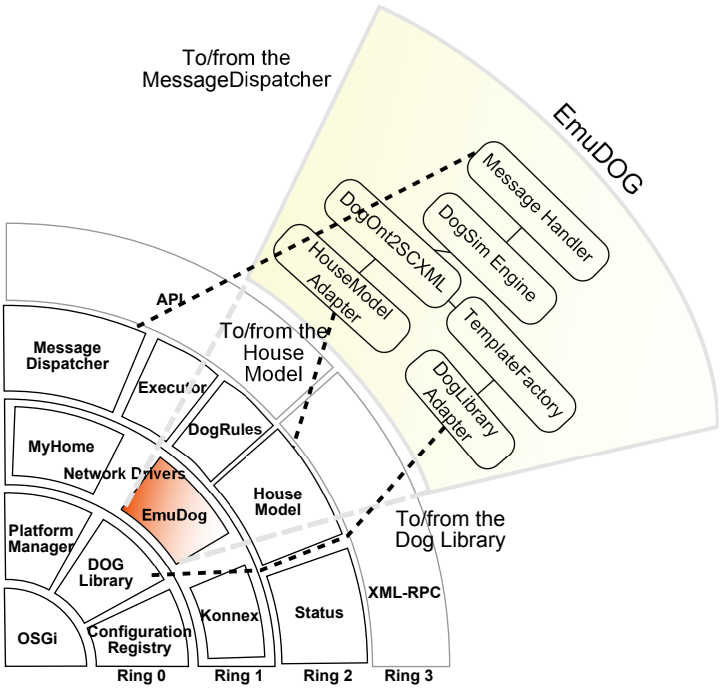


Figure 18: EmuDog.

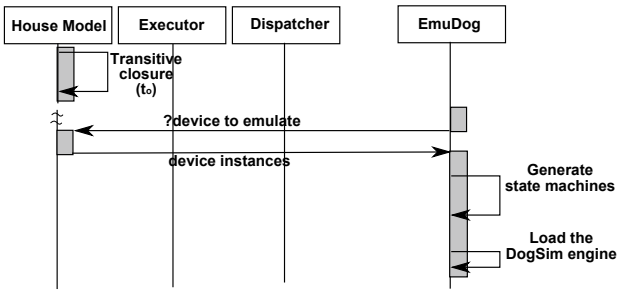


Figure 19: The EmuDog initialization sequence diagram.

Generated state machines are loaded in the DogSim Engine integrated in EmuDog and the bundle becomes ready.

At runtime (Figure 20), EmuDog works as any other Network Driver in Dog: it receives DogMessages requesting device actions (e.g., to switch a lamp on) and injects them as new events in the DogSim Engine. Running machines possibly change their states as a consequence of the injected event and detected state changes are then packed into one or more DogMessages and sent back to the MessageDispatcher bundle.

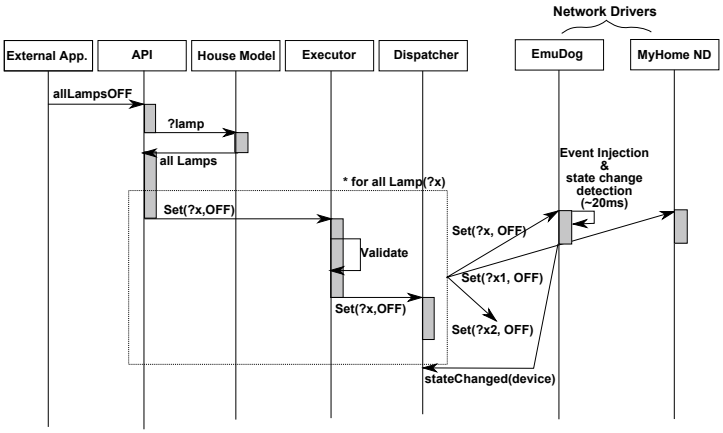


Figure 20: Sample EmuDog runtime sequence diagram.

8. Experimental Results

DogSim and EmuDog have been tested on two distinct environments: a synthetic, but realistic, case study involving a 6-room flat (called Simple Home) equipped with several domotic devices and a real building (Maison Equipée, Figure 21) composed of 3 domotic flats equipped with KNX home automation and located in St.Marcel, Valle d'Aosta, Italy.

Figure 21: Maison Equipée.

The first environment allowed to benchmark the framework performance in a rather controlled set-up, the second, on the other hand, enabled experimentation in a real-world setting.

8.1. Simple Home

The Simple Home is a synthetic environment composed of 6 rooms (see Figure 22) and counting 120 different domotic devices. The corresponding DogOnt description includes 378 class definitions and 53 different semantic relations for what concerns the ontology schema, and over 1400 concept instances describing specific devices, functionalities, states, commands and notifications.

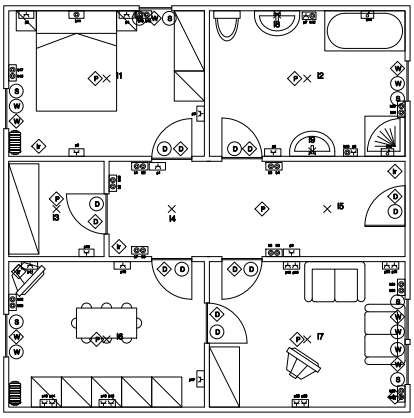


Figure 22: The Simple Home environment.

Simple Home devices are further divided into 25 real devices installed in two demo cases[10] (Figure 23) and 95 emulated devices.

Figure 23: The Demo Cases used in the experimental setup.

Emulated devices belong to a set of 11 different DogOnt classes, for which state diagram templates are available, with different diagram complexity (see Table 2).

While real devices are handled by the Dog MyHome and KNX network drivers, emulated devices are managed by EmuDog which takes care of both single-device simulation and device interconnection through the DogSim API. Connection handling (see Section 7), in particular, manages 3 different configurations that occur in the Simple Home test:

[10] Hosting BTicino MyHome and KNX devices, respectively.

Table 2: Device Templates

DogOnt class	#Inputs /#Outputs	#States	Timed transitions
Button	1/1	1	No
DoorActuator	4/0	5	Yes (4)
IRSensor	2/2	2	Yes (1)
Lamp	2/0	2	No
MainsPowerOutlet	2/0	2	No
ShutterActuator	5/0	5	Yes (2)
SimpleLamp	2/0	2	No
SmokeSensor	2/2	2	Yes (1)
Switch	2/2	2	No
ToggleRelay	1/2	2	No
WindowActuator	4/0	4	Yes (2)

- *Connections between real devices* connected to the same domotic plant are not managed by Dog, instead they already exists as part of the domotic network (wires and/or address associations).
- *Connections between real devices belonging to different networks* are managed by the Rules bundle already available in Dog, which enables cross-plant inter-operation [23].
- *Connections among emulated devices* as well as *connections between real and emulated devices* are managed through the EmuDog bundle.

The resulting emulation state diagram includes 95 device machines and 33 connectors, which are represented as orthogonal state diagrams counting a total amount of 384 defined states.

We measured elapsed times for model generation and model execution inside Dog, exploiting the House Model reasoning facilities (Jena OWL-Micro reasoner) for ontology-related tasks. In the model generation phase, two operations have been timed: the ontology loading and inference time t_o, measuring the time needed to load the DogOnt model of the SimpleHome and to compute the ontology closure; the template instantiation time t_i, measuring the average time required to instantiate all the templates associated to emulated devices.

Table 3 reports the corresponding figures, averaged over 10 test runs, obtained by running the tests on a double-core laptop PC, with an Intel Centrino2 P8400 processor and 4 GBytes of RAM[11]. CPU times needed for ontology loading/closure (House Model) and for template instantiation in EmuDog are comparable with the time required for the gateway startup and they are only needed at the beginning (one-shot operation). In particular, the template instantiation time t_i is around 6 s and is mainly due to the execution of SPARQL queries on the House Model. The highest time consumption contribution is due to the Jena API, which already proved to scale up to huge information bases (billion triples), much larger than the usual size of DogOnt building models (few thousand triples for very complex buildings).

Runtime execution of EmuDog is completely independent from ontology-related technologies. EmuDog employs less than 0.5 s for loading the 95 state machines in the Simple Home test. Event handling is very efficient and requires negligible times ($< 5\,\mathrm{ms}$) for unpacking Dog Messages and injecting them into the simulator engine, as well as for detecting machine state changes and for packing them back into Dog Messages. Timing of EmuDog-mediated interactions is comparable to, and sometimes faster than, timing of interaction between real devices

[11] Acer Aspire AS5930G, P8400, 4GBytes of RAM, 250GBytes HDD

Table 3: Model generation performance on the SimpleHome testset.

Time Interval	**Duration (s)**
t_o	6.15 (House Model)
t_i	6.36 (EmuDog)
Total generation time	12.51

managed through the MyHome and KNX drivers, thus allowing device emulation within the (weak) real time constraints required by smart environments. This result supports the adoption of EmuDog for emulation (HIL simulation) of IDE components and proves the effectiveness of the DogSim framework in simulating and emulating real-world complex IDEs like offices and smart buildings.

8.2. Maison Equipée

Maison Equipée is a home facility designed to help injured or people with disabilities to experiment with home assistive technologies including home automation solutions. It is owned by the public health administration of the Regione Valle d'Aosta and is mainly aimed at helping people with recent impairments to conceive, design and adapt their homes to their new conditions (i.e., to better tackle their physical impairments). The home is designed to offer medium term (three months in mean) residences allowing people to identify their specific home requirements and to live in a friendly and accessible home like setting while adapting their own homes. It is composed of 3 differently sized flats (Figure 24): two of them (Trait1 and 2) are designed for families and the remaining one for single patients (Maison Tech).

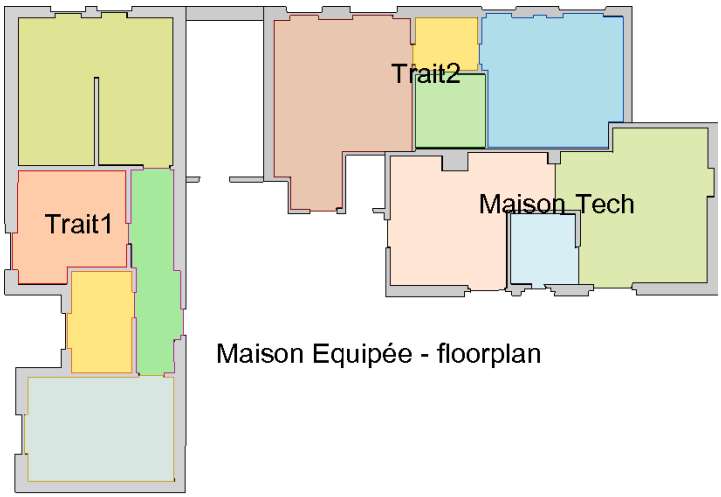


Figure 24: Simplified floorplan of Maison Equipée.

All the flats in Maison Equipée are instrumented with KNX home automation, each with different components to allow people to experiment with an appreciable variety of solutions on the market. Over 250 devices are installed in the three flats, divided as follows: 115 devices in

Trait1, 96 devices in Trait2 and 73 devices in Maison Tech. They include simple lamps, dimmer lamps, shutter, window and door actuators, infrared presence sensors, flooding detection systems, smoke alarms, etc.

We have been collaborating with the Maison Equipée management staff for over 6 months deploying different mixed configurations (real plus simulated devices) with the aim of designing and experimenting the next evolution of automation policies of the 3 flats. In particular, we exploited the DogSimg and EmuDog framework for supporting on-the-field testing of future configurations, e.g., simulating new actuators or new lamps controlled by real buttons, and for supporting home-to-user adaptation. In the latter case, we exploited the state machine infrastructure presented in this paper to allow reconfiguration of the environments responding to user needs. We experimented with home alarm reconfiguration exploiting connector machines to dynamically redirect alarms to either visual or auditive feedback depending on the user living in the home: visual feedback for deaf persons, auditory signals for blind people, combined auditory and visual alerts for families composed of normal and diversely able persons.

Although we've been experimenting for a couple of months, many open perspectives still need to be investigated and new application possibilities can be easily foreseen for Dog-powered environments. While achieving a quantitative and formal evaluation of the effectiveness of the proposed solution in the real-world is an open issue, we received very positive feedback from the Maison Equipée management staff, which was amazed from the variety of possible configurations that we have been able to experiment and evaluate together, without implementing a single change in the physical setting of the KNX home automation plant.

9. Conclusions

In this paper a framework for automatically generating and deploying state chart device simulators and emulators from ontology descriptions of domotic environments has been presented. The presented approach provides domotic system designers with both a powerful validation tool and an easy to deploy incremental development framework. The DogSim simulator and the DogEmu emulation bundle have been developed and integrated inside the Domotic OSGi Gateway (Dog), supporting the full range of possible emulation scenarios, from initial abstract design to final IDE deployment through Hardware In the Loop simulation. We tested the resulting platform on a synthetic, yet realistic, case study involving a 6-room flat and we deployed it in a real-world setting at the Maison Equipée. Evaluation involved performance benchmarking on a mixed emulation scenario with 25 real devices variously interacting with 95 emulated devices, and real-world experimentation with the Maison Equipée management staff. Results show that the approach is feasible and can easily address the complexity of real home environments. Future works will address deployment of the proposed approach in the real-world design work-flow, tackling open issues such as: effective interfaces, integration with currently existing CAD solutions, integrated development and automated deployment of designed solutions, etc.

References

[1] D. Bonino, F. Corno, DogOnt - Ontology Modeling for Intelligent Domotic Environments, in: A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin, K. Thirunarayan (Eds.), International Semantic Web Conference, number 5318 in LNCS, Springer-Verlag, 2008, pp. 790–803.

[2] 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.

[3] D. Bonino, F. Corno, Dogsim: A state chart simulator for domotic environments, in: Eighth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pp. 61–66.
[4] D. J. Cook, J. C. Augusto, V. R. Jakkulaa, Ambient intelligence: Technologies, applications, and opportunities, Pervasive and Mobile Computing Volume 5, Issue 4 (2009) 277–298.
[5] M. Jimenez, F. Rosique, P. Sanche, B. lvarez, A. Iborra, Habitation: A domain-specific language for home automation, IEEE SOFTWARE 26 (2009) 30–38.
[6] D. Alonso, C. Vicente-Chicote, O. Barais, V3Studio: A Component-Based Architecture Modeling Language, in: Engineering of Computer-Based Systems, IEEE International Conference on the, volume 0, IEEE Computer Society, Los Alamitos, CA, USA, 2008, pp. 346–355.
[7] G. Conte, D. Scaradozzi, A. Perdon, M. Cesaretti, G. Morganti, A simulation environment for the analysis of home automation systems, in: Proc. Mediterranean Conference on Control & Automation MED '07, pp. 1–8.
[8] K. Sycara, Multi Agent Systems, AI magazine 19(2) (1998) 79–92.
[9] G. Conte, D. Scaradozzi, Modeling and Control of Complex Systems, CRC Press, p. Ch.15.
[10] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8(3) (1987) 231–274.
[11] O. M. G. OMG, OMG Unified Modeling Language (OMG UML) v2.0, Technical Report, Object Management Group OMG, 2007.
[12] D. Bonino, F. Corno, Interoperation modeling for intelligent domotic environments, in: M. Tscheligi, B. de Ruyter, P. Markopoulus, R. Wichert, T. Mirlacher, A. Meschterjakov, W. Reitberger (Eds.), Ambient Intelligence, volume 5859 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2009, pp. 143–152.
[13] T. Saito, I. Tomada, Y. Takabatake, J. Ami, K. Teramoto, Home gateway architecture and its implementation, IEEE Transactiona on Consumer Electronics, 2000. ICCE. 2000 Digest of Technical Papers. 46 (2000) 194 –195.
[14] HGI, Home gateway technical requirements: Residential profile, Technical Report, Home Gateway Initiative, 2008.
[15] S.-L. Chung, W.-Y. Chen, MyHome: A Residential Server for Smart Home, Knowledge-Based Intelligent Information and Engineering Systems 4693/2007 (2007) 664–670.
[16] D. Winer, XML-RPC Specification, Technical Report, UserLand Software, 2003.
[17] D. Drusinsky, Modeling and Verification Using UML State Charts, Elsevier, 2006.
[18] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley Professional, 2005.
[19] S. Manesis, K. Akantziotis, Automated synthesis of ladder automation circuits based on state-diagrams, Advances in Engineering Software 36, Issue 4 (2005) 225–233.
[20] J. Barnett, R. Akolkar, R. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, State Chart XML (SCXML): State Machine Notation for Control Abstraction, Technical Report, W3C Working Draft, 2009.
[21] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: implementing the semantic web recommendations, in: WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, ACM, New York, NY, USA, 2004, pp. 74–83.
[22] B. Parsia, E. Sirin, Pellet: An OWL DL Reasoner, in: International Semantic Web Conference, p. (poster).
[23] D. Bonino, E. Castellina, F. Corno, Automatic Domotic Device Interoperation, IEEE Transactions on Consumer Electronics 55(2) (2009) 499–506.