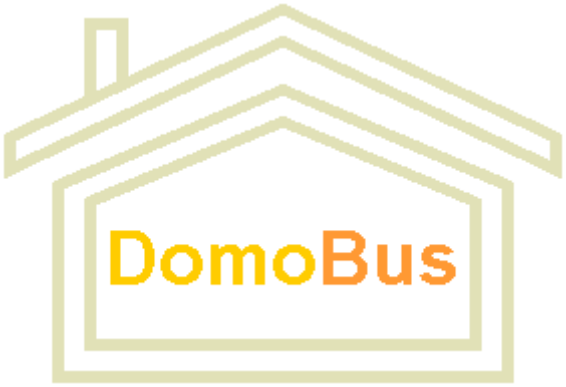
TÉCNICO LISBOA

DomoBus Simulator

Project Report



MEIC 2016/2017 – 2ND SEMESTER

AI - AMBIENCE INTELLIGENCE

Pedro Duarte – 78328

pedro.m.duarte@tecnico.ulisboa.pt

Abstract

This report describes the DomoBus Simulator tool, made for the Ambience Intelligence course, that simulates a DomoBus system and a user interaction with it. It covers the motivation behind the DomoBus system, as well as the motivation for a simulator in section 1. It will also present an overview of the architectures of DomoBus and of this till, in sections 2 and 3 respectively. Besides the overview, the tool's components will also be described in detail, as well as an in-depth analysis of the tools interface and implementation, and further discussion on the communication between the supervisor and the application.

Table of Contents

1. Introduction

1.1. Background and motivation

Home automation, or Domotics, is an area that focuses on the automation of tasks and combines technologies and services that aim to improve the quality-of-life of people, by making their houses:

- **More comfortable:** for example, automating lighting or heating control, door opening or scheduling appliances to turn on and off;
- **More secure:** detection of intruders or gas or water leaks;
- **More efficient:** automation of lighting and heating or other electronic appliances allow for a better energy management, resulting in economical savings and reduction of the ecological footprint [1];
- A **more laid-back** environment: because the house makes almost all the small decisions before they even think about them, the inhabitants can relax more and better without needing to pay attention to every little detail of the environment;
- Friendlier to **elder** or **disabled people**: for example, monitoring people and their medication intake, automating hard procedures for them, or warning relatives of a dangerous situation.

There are already a lot technologies that were born from the tentative standardization of domotic modules and communication, for example: **X10** [2] which is a less expensive than most alternatives and uses the power line as the communication medium; **KNX** [3] which was formerly known as EIB (European Installation Bus) and supports decentralized solutions, i.e. devices can interact directly with each other (P2P communication); and **CEBus** or **LonWorks**, that attempted to improve X10 functionality and reliability. However, because there are a lot technologies, potential domotic users can get confused as to which one to choose as these technologies are not interchangeable, i.e. they cannot communicate with each other. Thus, a user is forced to choose products from a single standard or company, which may not fulfill completely the user's preferences, or having incompatible modules in its house. Also, because most

technologies are complex to install and configure, it may be very hard for a user to change and adapt a system's functionality to its own preferences.

These problems were the motivation to the creation of **DomoBus** [4]: a technology developed as an academic project by Prof. Renato Nunes from Instituto Superior Técnico, which uses a simple and generic model to represent a domotic device (an appliance), has an XML specification language that enables the description of any system for any house and it's easy to translate to another specific technology. This makes the DomoBus able to interconnect all other existing technologies and ease the process of the system's adaption for the user. The DomoBus system will be further discussed in section 2 of this report.

The development of this project's subject, the DomoBus Simulator, was born from the problem of installation: once a user installs a module it's not that hard to remove it and place it somewhere else. However, as the complexity of the domotic system grows, so does the difficulty of configuration and installation, as well as the cost. Optimally, it would be best if the installation and configuration where to be made only one time.

The objective of this tool is to facilitate the process of decision for the user, simulating a real-word environment like its house, with every floor and division and their respective appliances, in a piece of software, helping the user easily change devices around without physically buying and installing them.

1.2. Objective

The objective of this project, subject of this report, was threefold:

- To create a simple and user-friendly graphical user interface that would be easy to use and is not cluttered by useless information;
- To develop a real, working tool that DomoBus users could use to help them materialize their ideal intelligent environment without actually buying the physical modules, allowing for the visualization of changes in the environment and helping in the process of decision when designing their own system;

- To create an application that, while missing some aspects outside the scope of simulation that are present in the DomoBus specification, would integrate well the current DomoBus system, being coherent with its modular approach and following strictly its specification language and protocols.

2. Related work

2.1. DomoBus

The DomoBus system is divided into two types of modules, as shown in figure 1: the Control Modules (CM) and the Supervision Modules (SM). Control Modules are comprised in the low-level components of the system, and are generally Arduino-like boards that connect to physical devices, i.e. sensors and actuators. An optional Router module (R) may also be added, that connects different DomoBus network segments.

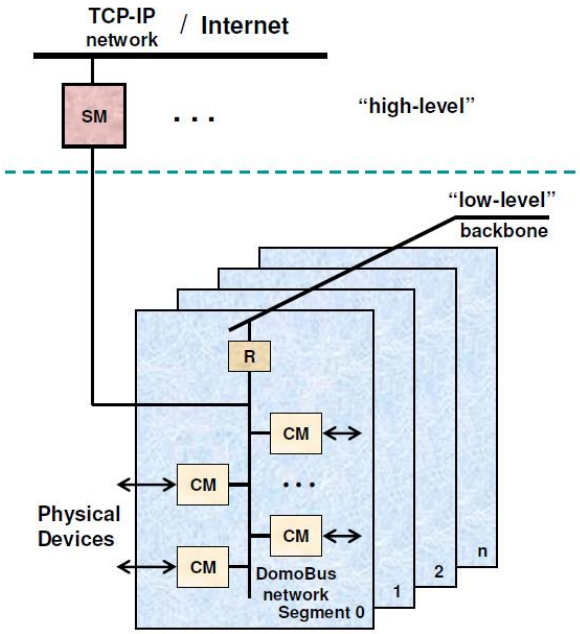


Fig. 1 - DomoBus Architecture

Supervision Modules are the high-level components of the system, and are computers or Raspberry-Pi like boards. These modules perform supervision tasks on the low-level components (receive and process information from sensors, and send commands to actuators), serve as an interface with users and may offer a remote access through an internet access point.

The goal of DomoBus is to offer a **modular** and **generic** platform for system specification that supports interchangeability with different technologies by, for example, allowing the development of generic applications that can be applied to any house, or allowing its users to personalize their system and program its behavior however they want.

This generic approach can be well observed in these three key aspects of the DomoBus solution:

- Definition of system **behavior**.
- Representation of a Domotic **device**;
- Definition of a system and a **house**;

All **system behavior** can be programable using: **rules**, which are simple if-then conditions; **timing actions**, which are actions that are executed after a specific delay; and **schedules**, which repeats an action at a certain time, that may be repeated with a specific period.

This generic approach can also be seen in the **device model**, as seen in in figure 2. DomoBus represents its devices by a collection of **properties**, and their interactions by three simple operations, which basically consist of reading and writing property values:

- GET: reads a property value;
- SET: changes a property value, which may imply an action over the environment;
- NOTIFY: autonomously notify a supervisor when a property value has changed, informing the new value.

Note that each device can belong to a **service** (for example, the Security service, which could group all alarms in a house) and that each Property can be of three types:

- Scalar, which is an integer (e.g. 0-100 value for a temperature in ºC);
- Enumerated, which is a pair designation-value (e.g. pairs Off = 0, On = 1);
- Array, that can be used to represent anything that cannot be represented by the previous types (e.g. a text display on an electronic board).

These three types of values are the only thing that is fixed in the specification.

Devices can also optionally be grouped in devices classes and is used as a means to organize device types. This option may prove very useful in a complex system with a lot of devices.

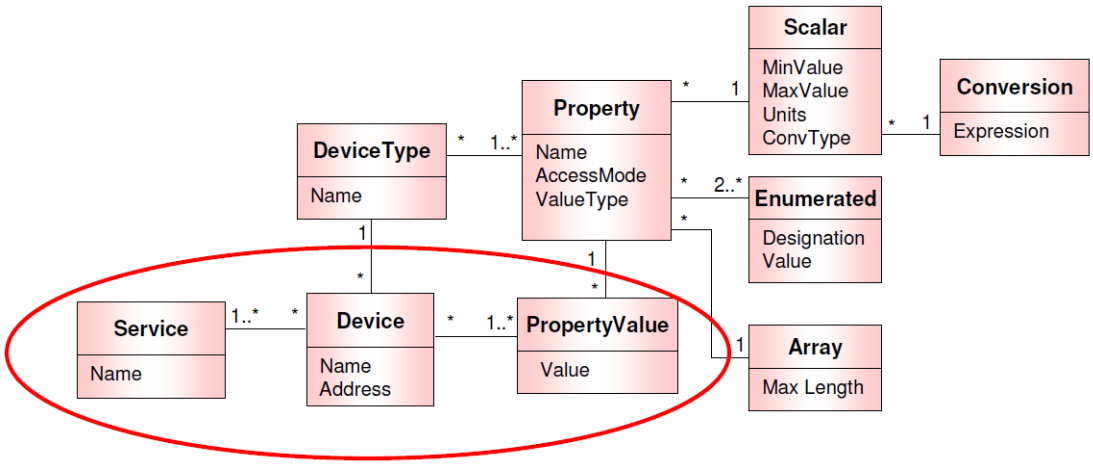


Fig. 2 - UML Model and instantiation of a device (in red)

Although this was not present in the earlier versions, a device can have one of two operation modes (OpMode): NORMAL, which when a value of a property changes, it keeps changed; and ACTION, that changes the value of a property, sends a signal to the supervisor, and then changes the value back to default: 0 for enumerated or scalar properties, and an empty string for array properties.

Finally, a house is hierarchically structured:

- a house may have one or more floors (ordered in function of their height);
- each floor may be divided into divisions;
- divisions may have devices, previously defined.

On the scope of the interface user-system, there is additional relevant information that a user could define:

- Favorites, which would allow a more direct access to some devices, divisions, etc.;
- Alarms, that would make sense to be displayed (in some system graphical interface) in a well-defined place given its major importance;

- Scenarios, which define a series of actions that would be executed in a specific scenario, for example, the scenario “Leaving the house” would have a list of actions that would turn off all the lights in the house when executed, instead of turning the off one by one;
- A state of the system can also be defined as well as saved, so that an application could quickly load it when it needed, for example in a testing context.

In this specification, users can also be defined with access levels. Because the devices also have access levels (one level for reading, one for writing), this introduces a simple access control mechanism: to perform an operation, the access level (corresponding to the operation) of the user must be higher.

The system specification is defined in XML [5], which specifies the house structure, the types of devices and devices existent, the behavior, etc. This specification language can be consulted in the DomoBus website, where a more in-depth explanation is provided.

2.2. DomoBus Supervisor

The DomoBus Supervisor is a program made by Prof. Renato Nunes that is responsible for mediating the communication between devices. It is a console application that takes a configuration file as its input and can execute commands, send messages to devices and receive messages back. It can also add schedules, sequences of actions, conditions or logical expressions to program the behavior of a given DomoBus system. The DomoBus Simulator may also communicate with the supervisor, resulting in a more realistic simulation experience. During development, the 64-bit version of the supervisor was used when testing the communication.

2.3. DomoBus Simulator: development environment

The DomoBus Simulator tool was developed in a Windows 10 64-bit environment, using Java 8 [6] (version 8u121 64-bit) as its main language, with the help of Eclipse [7] 64-bit IDE. As the focus of this tool is the user interface, I used Java Standard Widget Toolkit (SWT) [8] library, which provides and efficient and easy way to develop a graphical user

interface (GUI) with Java as a motor behind it. To ease the process of development of the graphical part of the interface even further, I also used Eclipse WindowBuilder [9] plugin, which makes possible to create and move around widgets without having to change the code manually.

3. DomoBus Simulator Architecture Overview

In this section, a brief overview of the tool's architecture will be provided, as well as an in-depth explanation of each of the individual modules that are part of the tool.

3.1. Description of the tool

The basic and mandatory input of this tool is a file containing a DomoBus specification (an XML file). When the user chooses a file, the DomoBus Simulator Back End (DMB) will then parse the configuration file and load the respective house and devices representations. These devices properties are then initialized with its respective default values. After that, the user is free to navigate in its house by choosing floors, divisions and devices, set any property value they want (limited to its properties restrictions) and observe how other devices react. The behavior of the system is not programmed in the tool, nor it is specified in the specification file: instead, it is programmed in the DomoBus Supervisor (DSupervisor), sending signals through the internet to change property values (SETs), depending on what it receives from the DMB (NOTIFYs). This communication is delegated to the DCommAPI, which is a wrapper to the DComm (DomoBus Communication) module, and mediates the communication between the application and the supervisor. The DComm receives a configuration file as input used to configure some variables that the protocol used, such as the destination and source IP and ports or messages timeouts. The user can also save and load the current state of the system in a file, which allows for a quick return to the workspace that was being previously used without needing to memorize or writing it down.

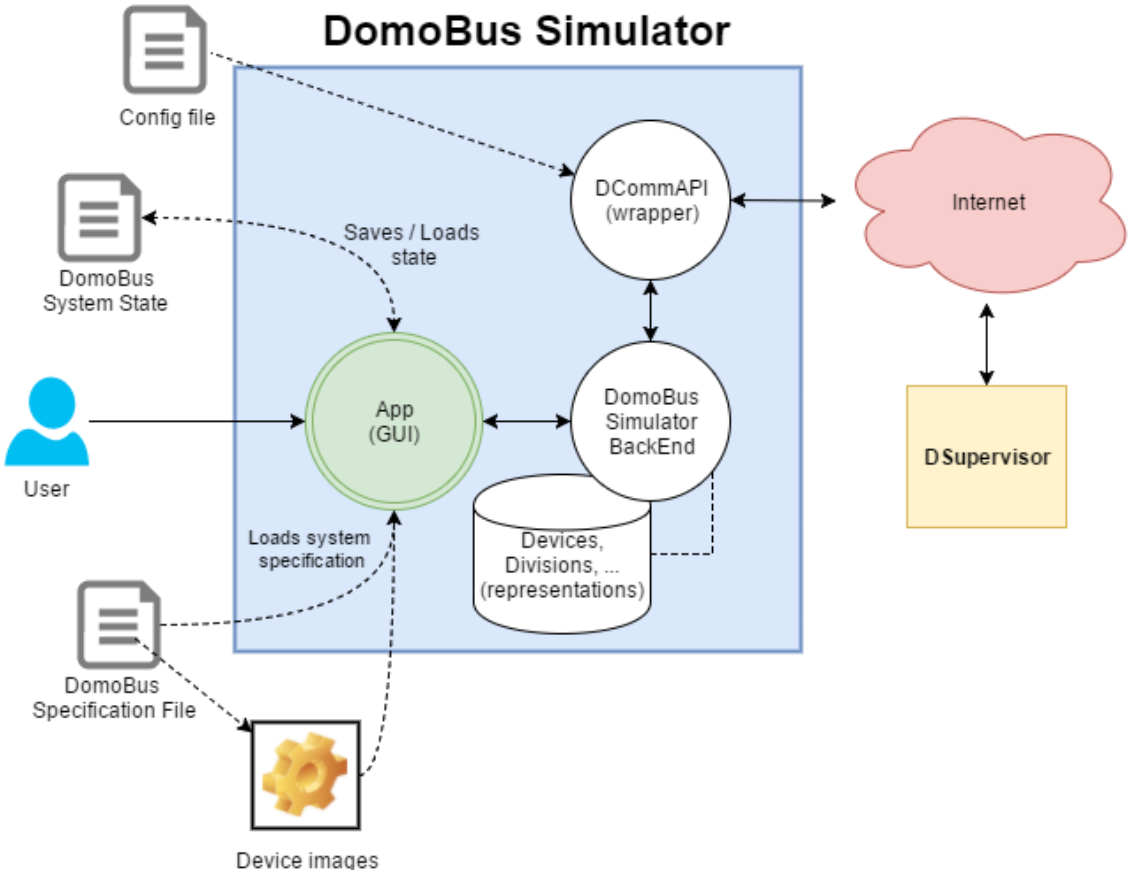


Fig. 3 - DomoBus Simulator Architecture

3.2. Tool components

3.2.1. Application

The **Application** is the module that contains the actual graphical user interface and is the only thing that the user sees and interacts with. It launches the DMB at the start, creates the interface contents (buttons, labels, …) and is then ready to interact with the user. Because this module is of paramount importance, it will be explained more in depth in the next section.

3.2.2. DomoBus Simulator Back End (DMB)

The **DomoBus Simulator Back End** (DMB) module is the core of this tool. This module is only initialized by the main application when a user loads a DomoBus system specification file. The DMB can be summarized by these four main functions:

- Loading the DomoBus System: the first job of this module is to read the XML file containing the system specification and parse it. This is done using the open-source library JDOM [10], which eases the process of parsing the file and seeking XML elements and attributes. While parsing it, the DMB will create the respective DomoBus elements and put them in memory.
- Answering to requests about DomoBus elements: give or change the property values the application asks for, as well as assert if some property is valid or invalid. For example, when a user wants to see a device, the application will ask the DMB the values of all the properties of that device and display the returned values in its interface. Likewise, when a user clicks a button that will change some property value of a certain device, the DMB will be asked by the application to change this device's corresponding property value. The change of values in this tool is done in two steps for the property types scalar and array. First, the user will change the value (by writing another string in the array, or altering the displayed value in a scalar); this will trigger a procedure in the DMB that will keep record of the properties that the user is trying to change, i.e. the tentative values. Then, when a user is satisfied with the values, they will press another button that will commit the changes, triggering another DMB procedure that will notify the supervisor of these new changes and clear that property of the tentative values record, finally returning the new values to the interface. For the enumerated case, the value changing is done in one step because of the simpler method of changing in the interface.
- Loading and saving the system state: optionally, a user can save the current state of the application, i.e. all the devices and their respective property values, as well as their validity can be saved in a file for posterior usage. The saving is done by writing the current values off all the devices that are currently loaded in a XML file, which from this point on will be treated as the system state file (the format of the content of this file will be explained more in depth in section 4). Then, when a user wants to restore a previous state of the system, he just needs to load the file containing the system state. This is done by the DMB which parses this XML file and assumes that the correct system specification is already loaded. Then, the values of current loaded system will simply be replaced by those

previously saved. Note that the user needs to load the correspondent specification file of the system beforehand. For instance, if a user loads a specification file of system A, it can then only load the state of system A, or else the result will be unpredictable, as to preserve the flexibility that is available on the DomoBus environment there is no clean way to make sure that the correct files are being loaded.

- Serving as the mediator between the DComm module and the interface: when a value is changed, the tool must notify the supervisor of the new value, but this is done through the DMB, that tells the DComm module to send a notify to the supervisor with the device address, property descriptor and the new value of said property that has been changed. The device address is simply an integer directly loaded from the device instantiation on the system specification file. The property descriptor is an integer that contains information about the property id, the validity of said property and the type of the property. These three pieces of information are joined together by a bitwise OR, which allows to send all the relevant information about the property in a single integer, avoiding the transmission of unnecessary arguments.

3.2.3. DCommAPI

The DComm module is responsible for handling the communication between some DomoBus application and a DomoBus supervisor. This module was already implemented in C, but because the DomoBus Simulator is made using Java, a wrapper was necessary in order to properly reuse what was already done. The only job of this wrapper is to call the functions on the original API by using the Java Native Interface [11] (JNI), which enables java code running in a Java Virtual Machine to interact (call and be called) with native applications or libraries written in other languages. The native DCommAPI is initialized by providing a configuration file of the same type that the supervisor takes as an input, and similarly, is used to point to the other application that it is supposed to communicate with (in this case, the supervisor). After being initialized, the module can now send and process messages (GETs, SETs and NOTIFYs).

4. Details of the solution

This section will give a more in-depth description of the graphical user interface, as well as provide a deeper explanation of the process of communicating with the supervisor.

4.1.1. Graphical User Interface

When opening the application, the user will be presented with a main window, as shown in figure 4. This main window is divided into three main sections: the workspace section, the DomoBus house section, and the properties section. All of these sections are empty until the user chooses a DomoBus specification file to load. This can be done by selecting the “File” menu and then the “Open DomoBus specification file…” option, as shown in figure 5. By doing this, the DMB will read and parse the file and load the respective system specification, including the house structure and all its devices.

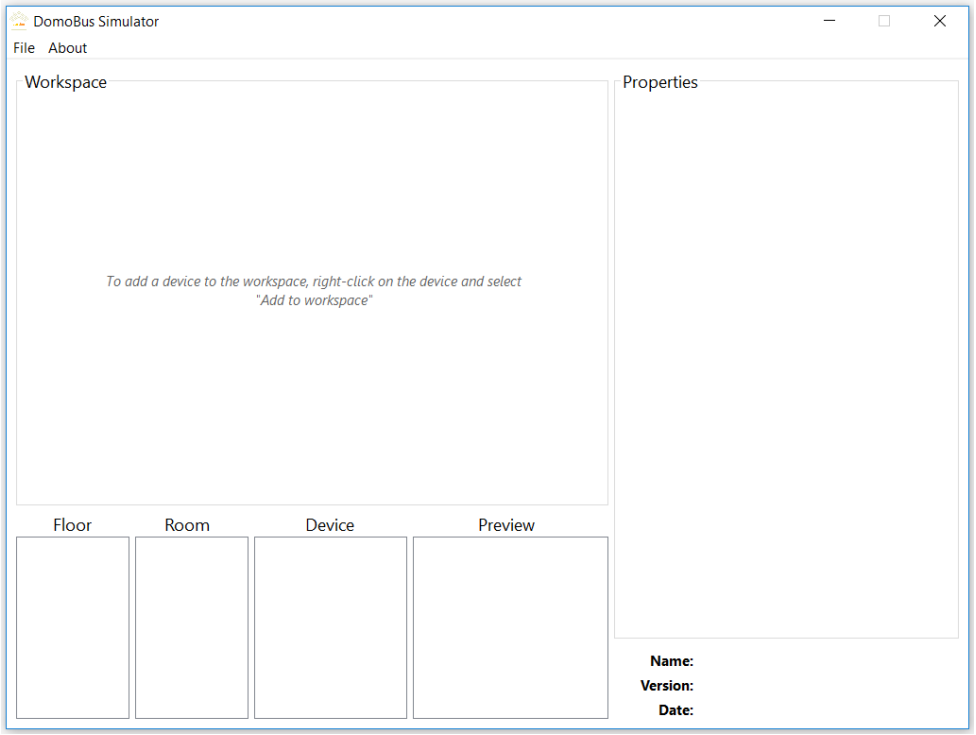


Fig. 4 – Main window

This will fill the labels in the bottom right of the window, which describe the DomoBus system specification metadata, and will also fill the DomoBus house section which now contains a hierarchically navigable representation of the house physical structure. Selecting the floor will show all rooms of that floor in the “Room” subsection, and

selecting a room will display all the devices in that room in the “Device” subsection. Furthermore, by selecting a device, a preview of the properties of that device can be seen in the “Preview” subsection, seen in figure 6. This provides a quick reminder of what are the properties of a device and their respective current values.

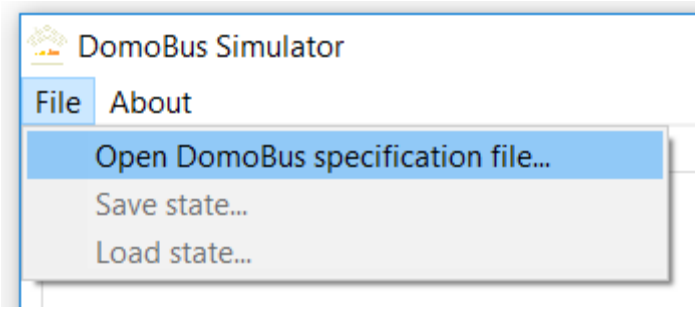


Fig. 5 - File loading menu

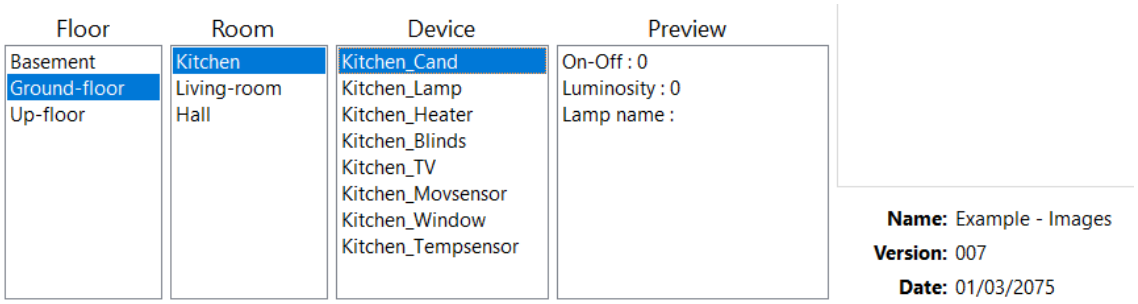


Fig. 6 - House navigation and property preview

To actually change the value of a device property, a user needs to add a device to the workspace section. To do this, a user needs to right-click the device he wants to add to the workspace and select the option “Add to workspace”. This will add a graphical representation of the device in the workspace section, and can be done with multiple devices as shown in figure 7.

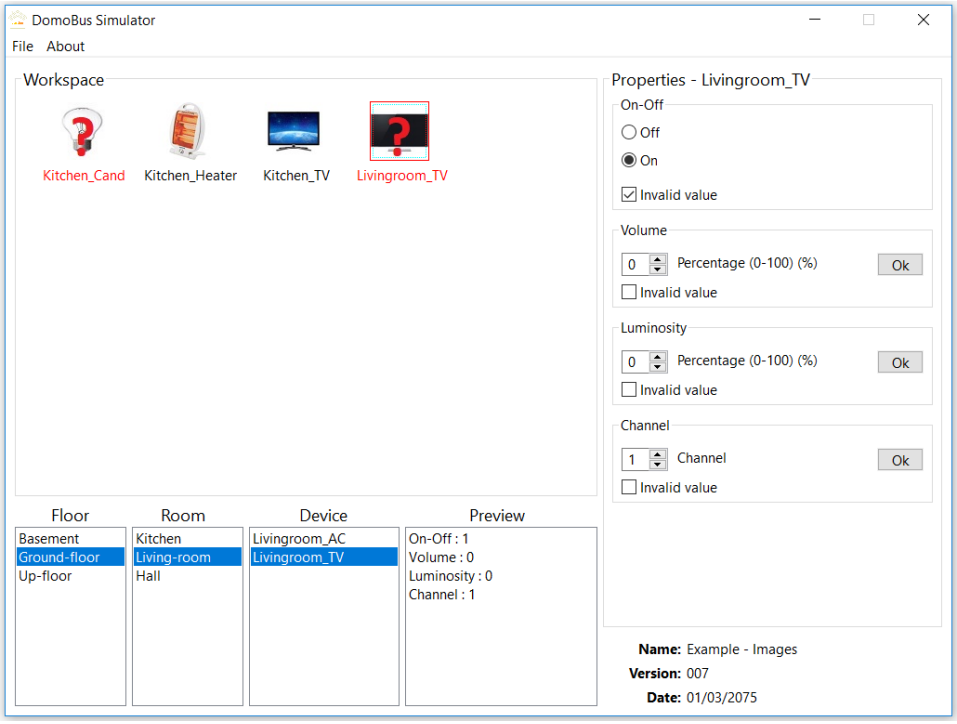


Fig. 7 - Workspace with images

Note that these graphical representations must be provided by the user and pointed to in the DomoBus specification file, in the ImageList section, as it can be seen in figure 8. The way the simulator selects which image to display in the workspace is: for each property, checks which is the last (in the order defined in the specification file) that has a value in a defined range, selecting that one. The images, however, are limited to a 62x62 pixels, as they could not be scaled down. At any time, a user can remove a device from the workspace by right-clicking the device in the workspace and then selection the option “Remove from workspace”. The workspace is a very useful tool because it helps to visualize a context that the user wants to work with. For example, if a user wants to see the behavior of all the lamps in the house when executing a certain action, instead of looking through all the devices in the bottom section manually while traversing the house they can, instead, add all the lamps to the workspace and see what happens to them all at once.

```
<DeviceType Description="-" ID="5" Name="Heater" RefDeviceClass="2" ImageDefault="Images/Heater_Default.jpg">
    <PropertyList>
        <Property AccessMode="RW" ID="1" ImagePath="Images/OnOff.jpg" Name="Heater Commands" OpMode="NORMAL" RefValueType="2" ValueType="ENUM"/>
        <Property AccessMode="RW" ID="2" ImagePath="Images/Temperature.jpg" Name="Temperature" OpMode="NORMAL" RefValueType="4" ValueType="SCALAR"/>
        <Property AccessMode="RW" ID="3" ImagePath="Images/Motor.jpg" Name="Power" OpMode="NORMAL" RefValueType="2" ValueType="SCALAR"/>
    </PropertyList>
    <ImageList>
        <Image ID="1" RefProperty="1" ValueRange="0-0" ImagePath="Images/Heater_Off.jpg"/>
        <Image ID="2" RefProperty="1" ValueRange="1-1" ImagePath="Images/Heater_Cooling.jpg"/>
        <Image ID="3" RefProperty="1" ValueRange="2-2" ImagePath="Images/Heater_Heating.jpg"/>
    </ImageList>
</DeviceType>
```

Fig. 8 - Image pointing in the specification file

Once all the needed devices are in the workspace, a user can then change whatever device's property values they want, by clicking on a device in the workspace. The properties section will be populated with some subsections, one for each property that the device has. To change a property of the enumerated type, a user simply needs to click the button correspondent to the new value. To change a property of the scalar type, they can either write the new value in the number box or click on the arrows, then when the correct value is displayed, press the button "Ok" to commit the changes. To change an array, a user needs to write, on the text box, the new value, and then press the button "Ok". Note that each property subsection has a box displaying the label "Invalid value", representing that the supervisor does not know the value of a specific property. This may happen for a certain number of reasons, e.g. a message was in an incorrect format, the message was lost in the network before reaching the supervisor, etc., and will cause the device to which the property belongs also be marked as invalid. In this case, the device's image used will be the default image, also defined in the specification file.

Finally, once the user is done changing values around, they can save the current state of the simulator by selecting the option "Save state…" under the "File" menu; similarly, if they wish to load a previous state, they can do so by selecting the option "Load state…". An example of the XML file containing the state of a previous usage is shown in figure 9, displaying the saved state of the example used in this report, in figure 7. Note that the current workspace is also saved, as well as the validity of each device.

```
<?xml version="1.0" encoding="UTF-8"?>
<Components>
  <DeviceStateList>
    <DeviceState InvalidValue="True" RefDevice="1" RefProperty="1" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="1" RefProperty="2" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="1" RefProperty="3" Value="" />
    <DeviceState InvalidValue="False" RefDevice="2" RefProperty="1" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="2" RefProperty="2" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="2" RefProperty="3" Value="" />
    <DeviceState InvalidValue="False" RefDevice="3" RefProperty="1" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="3" RefProperty="2" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="3" RefProperty="3" Value="10" />
    <DeviceState InvalidValue="False" RefDevice="4" RefProperty="1" Value="2" />
    <DeviceState InvalidValue="False" RefDevice="4" RefProperty="2" Value="10" />
    <DeviceState InvalidValue="False" RefDevice="4" RefProperty="3" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="5" RefProperty="1" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="5" RefProperty="2" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="6" RefProperty="1" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="6" RefProperty="2" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="6" RefProperty="3" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="6" RefProperty="4" Value="1" />
    <DeviceState InvalidValue="False" RefDevice="7" RefProperty="1" Value="0" />
    <DeviceState InvalidValue="True" RefDevice="8" RefProperty="1" Value="1" />
    <DeviceState InvalidValue="False" RefDevice="8" RefProperty="2" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="8" RefProperty="3" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="8" RefProperty="4" Value="1" />
    <DeviceState InvalidValue="False" RefDevice="9" RefProperty="1" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="9" RefProperty="2" Value="0" />
    <DeviceState InvalidValue="False" RefDevice="10" RefProperty="1" Value="10" />
  </DeviceStateList>
  <WorkspaceList>
    <Device ID="1" />
    <Device ID="4" />
    <Device ID="6" />
    <Device ID="8" />
  </WorkspaceList>
</Components>
```

Fig. 9 - Saved state example

4.1.2. DCommAPI – Supervisor communication

Because the simulator does not use the DComm module directly, but instead uses a wrapper, it is important to explain how the actual wrapper works. The wrapper basically mimics the commands given to it to the actual native API. On initialization, the wrapper initializes the API, on sending a message, the wrapper asks the API to send a message, etc. To do this, the native library is loaded by calling the system method:

```
System.loadLibrary(DCOMM_DLL);
```

This method loads the library specified in the argument; in this case, it will load a library that is present that is called DComm.dll. This dll file is the product of the compilation of the C code that implements the header file generated by the JNI, as per result of the following steps [12]:

1. Declare the native methods in the Java code; an example is shown in figure 10;
2. Compile the java class, then create the C/C++ header file: DComm_API.h, by executing the commands "*javac DCommAPI.java*" and "*javah DCommAPI*". The result will be something like the example in figure 11.
3. Implement and compile the API in C. In this project, the API came from Prof. Renato Nunes, so I was given the dll file already containing the compiled library, ready to use.

```
private native int DComm_config_from_file(String filename);

private native int process_msg_get(int dev_addr, int prop_desc,
                                   byte[] value_ptr, int value_invalid_p);
```

Fig. 10 - Native method declaration

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class DComm_DCommAPI */

#ifndef _Included_DComm_DCommAPI
#define _Included_DComm_DCommAPI
#ifdef __cplusplus
extern "C" {
#endif
#undef DComm_DCommAPI_DCOMM_OK
#define DComm_DCommAPI_DCOMM_OK 0L
/*
 * Class:     DComm_DCommAPI
 * Method:    DComm_config_from_file
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_DComm_DCommAPI_DComm_1config_1from_1file
  (JNIEnv *, jobject, jstring);
```

Fig. 11 - C/C++ header file

This API can now be called from another place, like the DMB does. For example, to notify the supervisor of a change of a device's property, the DMB calls the method sendNotify() of the wrapper, giving as an argument the device address, the property descriptor a byte array containing the new value. Then, the wrapper simply calls the corresponding method of the native API.

5. Evaluation

To perform a continuous evaluation during development, I used the supervisor provided by Prof. Renato Nunes, as it served as a powerful tool to verify if everything in the simulator is working as expected, as well as perform some tests without having to use

real devices. Overall, there are no performances issues, except the ones caused by the communication over the network; because the network is, in general, not reliable, there may be the need to wait for a timeout, or simply because the connection may be slow. However, because of my lack of experience designing interfaces, the resulting code may not be the best overall, as I had to balance code efficiency, interface clarity and respect the specifications of the DomoBus domain.

6. Conclusion

Overall, the objective of developing a usable tool was achieved, as well as producing a clear graphical interface, while still respecting the modularity and flexibility of a DomoBus system specification. Some future improvements might include:

- A script that allows for automated action executing in the simulator. This script would be a list of actions on some DomoBus devices (SETs) and an optional delay that when loaded by the simulator would execute them in sequence. This would be a more realistic and convenient way of simulating a scenario, e.g. arriving at home, which would simulate the user passing through some divisions;
- More flexibility with respect to the number of devices and properties allowed in the graphical interface: right now, the workspace has a maximum number of devices that can be shown at once, and so does the properties section. It would be interesting to add a tab system for more complex DomoBus environments.

7. References

Figures 1 and 2 were borrowed from Ambience Intelligence class slides.

Cover picture was borrowed from the DomoBus website: www.domobus.net

[1] Ecological footprint - https://en.wikipedia.org/wiki/Ecological_footprint

[2] X10 - [1] https://www.x10.com/

[3] KNX - https://www.knx.org/knx-en/index.php

[4] (Renato Nunes) *DomoBus – A New Approach to Home Automation*, http://www.domobus.net/papers/03-CLEEE03.pdf

[5] Further DomoBus related papers - http://domobus.net/papers/

[6] Java - www.java.com

[7] Eclipse - www.eclipse.org/

[8] Java SWT - www.eclipse.org/swt/

[9] Eclipse WindowBuilder - www.eclipse.org/windowbuilder/

[10] JDOM - http://www.jdom.org/

[11] JNI Specification - http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

[12] JNI tutorial - https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html