Application for Defining Complex Smart Home Behaviors

Ricardo Morais Lupi Nogueira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Renato Jorge Caleira Nunes
Prof. Paulo Rogério Barreiros D’Almeida Pereira

Examination Committee
Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Renato Jorge Caleira Nunes
Member of the Committee: Prof. Miguel Filipe Leitão Pardal

November 2019
Declaration

I declare that this document is an original work of my own authorship and that it fulfils all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgements

Writing this thesis was only possible due to the support and availability of multiple people, to which I would like to thank.

To professor Renato Nunes, for his excellent work as a supervisor, the long brainstorming hours spent in his office and for the support during all stages of the work.

To my parents, who were always present in this long journey, for the unconditional support and care they always gave me. To my grandmother, who always pushed me to do better and for all her advice.

To Ana, for being there at the most complicated moments and for her genuine support.
Abstract

The acclaim of home automation is steadily increasing due to the IoT trend and the availability of cloud services for streamed data extraction and event processing for swarms of devices. Currently, there is a heterogenous landscape of home automation standards and protocols based in local networks, which hinder standardization and integration processes of domotics. Cloud solutions bridge the gap between multi-vendor integrations, but still lack SaaS offerings to build and manage interesting and complex behaviors for super automated homes, without technical know-how. Available SaaS house behavior builder applications provide over simplistic features for ruleset management, typically offering just simple cause and effect chains (if-then-else). Due to this problem, the DomoBus system and its device and supervision models were used as flexible and generic resources to conceive the DomoBus Automation Block (DAB) and build the DomoBus House Manager (DHM) program. The DHM empowers the common user to create non-trivial behaviors with ease and provides important tools such as management of state machines and DAB aggregation. DABs are completely agnostic to device vendors, allowing hundreds of devices to be instantiated in the final house DAB whilst the end user is completely abstracted of the underlying behavior specificities.
# Table of Contents

List of Acronyms ........................................................................................................... vii

Table of Figures ............................................................................................................. ix

1. **Introduction** ............................................................................................................. 1
   1.1. Motivation ............................................................................................................ 1
   1.2. Objectives .......................................................................................................... 3
   1.3. Document Structure .......................................................................................... 3

2. **State of the Art** ....................................................................................................... 4
   2.1. Home Automation Concepts ............................................................................. 4
       2.1.1. Home Automation Systems Overview ....................................................... 4
       2.1.2. Rule-Based vs User-Controlled Automation ............................................ 6
       2.1.3. Home Automation Standards ................................................................. 7
       2.1.4. Super Automated Homes ......................................................................... 11
   2.2. Cloud-Based Home Automation Solutions ....................................................... 13
       2.2.1. Muzzley ...................................................................................................... 13
       2.2.2. Samsung SmartThings ............................................................................ 17
       2.2.3. IFTTT ......................................................................................................... 22

3. **DomoBus** .............................................................................................................. 24
   3.1. Overview ............................................................................................................. 24
   3.2. System Architecture ......................................................................................... 25
       3.2.1. Control Layer .......................................................................................... 25
       3.2.1. Supervision Layer ................................................................................... 28
   3.3. Device Model ..................................................................................................... 31
   3.4. Supervision Model ........................................................................................... 32
   3.5. Supervision Library .......................................................................................... 34

4. **Developed Work** ................................................................................................... 39
   4.1. DHM Concepts ................................................................................................. 39
       4.1.1. DomoBus Automation Blocks ..................................................................... 39
       4.1.2. Protected Outputs and Critical Services ................................................... 47
       4.1.3. Messages to External Applications ............................................................ 47
       4.1.4. DHM Behavior Parameterization ............................................................... 48
       4.1.5. Behavior Distribution ............................................................................... 49
4.2. DHM Composition .................................................................................. 53
  4.2.1. Introduction ................................................................................... 53
  4.2.2. Top Navbar .................................................................................. 53
  4.2.3. Overview Tab .............................................................................. 54
  4.2.4. Rooms Tab .................................................................................. 55
  4.2.5. Devices Tab ............................................................................... 56
  4.2.6. Messages Tab ............................................................................ 57
  4.2.7. Parameters Tab .......................................................................... 58
  4.2.8. Behavior Builder Tab ................................................................. 58
  4.2.9. Aggregator Tab ......................................................................... 63

4.3. DHM Implementation ......................................................................... 64
  4.3.1. Software Architecture ............................................................... 64
  4.3.2. DAB XML File Structure ............................................................ 68
  4.3.3. Assembly File Structure ............................................................. 70

5. Results Analysis .................................................................................... 71
  5.1. Use Case ......................................................................................... 71
  5.2. Qualitative Evaluation .................................................................. 78

6. Conclusion ............................................................................................ 79

References ............................................................................................... 80

Annexes .................................................................................................... 82
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Alternating Current</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APO</td>
<td>Application Object</td>
</tr>
<tr>
<td>APS</td>
<td>Application Support Sublayer</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>CM</td>
<td>Control Module</td>
</tr>
<tr>
<td>CSMA/CA</td>
<td>Carrier Sense Multiple Access with Collision Avoidance</td>
</tr>
<tr>
<td>DAB</td>
<td>DomoBus Automation Block</td>
</tr>
<tr>
<td>DCN</td>
<td>DomoBus Control Network</td>
</tr>
<tr>
<td>DCOMM</td>
<td>DomoBus Communication (Library)</td>
</tr>
<tr>
<td>DDNS</td>
<td>Dynamic Domain Name Service</td>
</tr>
<tr>
<td>DHM</td>
<td>DomoBus House Manager</td>
</tr>
<tr>
<td>ETS</td>
<td>Engineering Tool Software</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>I/O</td>
<td>Input-Output</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Provider</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MQTT</td>
<td>Message Queuing Telemetry Transport</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>PAN</td>
<td>Personal Area Network</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RBAC</td>
<td>Role Based Access Control</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SM</td>
<td>Supervision Module</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UPnP</td>
<td>Universal Plug and Play</td>
</tr>
<tr>
<td>USART</td>
<td>Universal Synchronous Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless Local Area Network</td>
</tr>
<tr>
<td>WPAN</td>
<td>Wireless Personal Area Network</td>
</tr>
<tr>
<td>ZC</td>
<td>Zigbee Coordinator</td>
</tr>
<tr>
<td>ZDO</td>
<td>Zigbee Device Object</td>
</tr>
<tr>
<td>ZED</td>
<td>Zigbee End Device</td>
</tr>
<tr>
<td>ZR</td>
<td>Zigbee Router</td>
</tr>
</tbody>
</table>
Table of Figures

Figure 1 – Capabilities and possible interconnections between embedded systems .............................................. 4
Figure 2 - Basic living room IoT node placement example ...................................................................................... 12
Figure 3 – Defining a Muzzley agent on its mobile application UI [25] ................................................................. 13
Figure 4 - Connections between Muzzley’s Cloud (selfcare channel), the manufacturer’s manager (gateway) and the manufacturer’s cloud or device [25] ................................................................. 14
Figure 5 - Muzzley’s device hierarchy, the breakdown of a device [25] ................................................................. 14
Figure 6 - SmartThings system [26] .................................................................................................................. 18
Figure 7 - Issuing commands to a device via Mobile Application or Automation on SmartThings [26] .......... 20
Figure 8 - Device state change initiated by third-party cloud or mobile application / Automation (modified) [26] ............................................................................................................................................ 20
Figure 9 – Screen flow of a user creating a custom applet on the IFTTT mobile application .................. 22
Figure 10 - DomoBus architecture - Control Network emphasis ........................................................................... 26
Figure 11 - Structure of a Device Address (32 bits), used when interacting with DCN [28] .............. 27
Figure 12 - DomoBus architecture – Supervision Network emphasis ............................................................... 29
Figure 13 - List of Device Classes on the DomoBus System Specification XML file .......................... 36
Figure 14 - List of Conversion Formulas & Objects on the DomoBus System Specification XML file .. 36
Figure 15 - List of Value Types on the DomoBus System Specification XML file ........................................ 37
Figure 16 - List of Device Types on the DomoBus System Specification XML file ........................................ 37
Figure 17 - List of Access Levels on the DomoBus System Specification XML file ................................ 37
Figure 18 - List of Users on the DomoBus System Specification XML file ..................................................... 37
Figure 19 - House Structure on the DomoBus System Specification XML file ........................................... 38
Figure 20 - List of Services on the DomoBus System Specification XML file .............................................. 38
Figure 21 - List of Instanced Devices on the DomoBus System Specification XML file ............................... 38
Figure 22 - List of Devices States on the DomoBus System Specification XML file ..................................... 38
Figure 23 - List of Scenarios on the DomoBus System Specification XML file .............................................. 38
Figure 24 - Example of Expandable Input ........................................................................................................ 40
Figure 25 - A state machine with three states and their transitions to showcase automatic enables/disables ................................................................................................................................................. 42
Figure 26 - Reference architecture for the basis DAB ...................................................................................... 43
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Example of DAB Fusion</td>
</tr>
<tr>
<td>28</td>
<td>Example of DAB Union</td>
</tr>
<tr>
<td>29</td>
<td>Resulting Graph and Maximum Spanning Tree connections in bold</td>
</tr>
<tr>
<td>30</td>
<td>Adjacency Matrix of the Example Graph</td>
</tr>
<tr>
<td>31</td>
<td>Dendrogram output of the Single Link Hierarchical Agglomerative Clustering algorithm applied to the Maximum Spanning Tree of the Example Graph</td>
</tr>
<tr>
<td>32</td>
<td>Popup to fix disconnected inputs and outputs at house behavior generation time</td>
</tr>
<tr>
<td>33</td>
<td>Overview tab of the DHM</td>
</tr>
<tr>
<td>34</td>
<td>Rooms tab of the DHM</td>
</tr>
<tr>
<td>35</td>
<td>Devices tab of the DHM</td>
</tr>
<tr>
<td>36</td>
<td>Messages tab of the DHM</td>
</tr>
<tr>
<td>37</td>
<td>Parameters tab of the DHM</td>
</tr>
<tr>
<td>38</td>
<td>Behavior Builder tab, Properties sub-tab of the DHM</td>
</tr>
<tr>
<td>39</td>
<td>Aggregator tab of the DHM</td>
</tr>
<tr>
<td>40</td>
<td>Example of basis DAB XML</td>
</tr>
<tr>
<td>41</td>
<td>Example of aggregated DAB XML</td>
</tr>
<tr>
<td>42</td>
<td>Example of final DAB XML</td>
</tr>
<tr>
<td>43</td>
<td>Example of Assembly file</td>
</tr>
<tr>
<td>44</td>
<td>Family house case study – Outdoor view</td>
</tr>
<tr>
<td>45</td>
<td>Outdoor Scenario 1 built on DHM Behavior Builder tab, properties sub-tab</td>
</tr>
<tr>
<td>46</td>
<td>Outdoor Scenario 1 built on DHM Behavior Builder tab, states sub-tab</td>
</tr>
<tr>
<td>47</td>
<td>Family house case study - Living Room view</td>
</tr>
<tr>
<td>48</td>
<td>Living Room Scenario built on DHM Aggregator tab</td>
</tr>
<tr>
<td>49</td>
<td>Family house case study - Kitchen view</td>
</tr>
<tr>
<td>50</td>
<td>Kitchen and Living Room Interconnections</td>
</tr>
<tr>
<td>51</td>
<td>Instantiation of Hardware Devices on the Rooms’ Inputs and Outputs</td>
</tr>
<tr>
<td>52</td>
<td>Web view interface editor at the selfcare page</td>
</tr>
<tr>
<td>53</td>
<td>DomoBus Control Network Packet Format [28]</td>
</tr>
<tr>
<td>54</td>
<td>A) Example of a Fan Speed capability definition in SmartThings; B) JSON POST request body for sending a command to a SmartThings device with a Switch capability [26]</td>
</tr>
<tr>
<td>55</td>
<td>Structure of the Data field on the DCN packet for the GET and SET message types [28]</td>
</tr>
</tbody>
</table>
Figure 56 - Structure of the Data field on the DCN packet for the NOTIFY message type [28]............. 84
Figure 57 - Structure of the Property Descriptor byte on the DCN and Supervision Level packet Data fields [28]........................................................................................................................................ 84
Figure 58 - Structure of the Data field on the DCN packet for the EXEC message type [28] ............ 84
Figure 59 – DomoBus Supervision Level Packet Format [28] ................................................................ 84
Figure 60 – Structure of the Data field on Supervision Level packet for GET message type [28]....... 85
Figure 61 – Structure of the Data field on Supervision Level packet for the A_GET message type [28] ........................................................................................................................................ 85
Figure 62 – Structure of the Data field on Supervision Level packet for the SET message type [28] .. 85
Figure 63 – Structure of the Data field on Supervision Level packet for the NOTIFY message type [28] ........................................................................................................................................ 85
Figure 64 – Structure of the Data field on Supervision Level packet for the EXEC message type [28] 85
Figure 65 - Kitchen's Scenario 1 DAB ................................................................................................... 85
Figure 66 - Kitchen's Scenario 2 DAB ................................................................................................... 85
Figure 67 - Kitchen's Scenario 3 DAB ................................................................................................... 86
Figure 68 - Kitchen's Scenario 4 DAB ................................................................................................... 86
Figure 69 - Kitchen's Scenario 5 DAB ................................................................................................... 86
Figure 70 - Living Room's Scenario 1 and 2 DAB .................................................................................. 87
Figure 71 - Living Room's Scenario 3 DAB ............................................................................................ 88
Figure 72 - Living Room's Scenario 4 DAB ............................................................................................ 88
Figure 73 - Living Room's Scenario 5 DAB ............................................................................................ 89
Figure 74 - Living Room's Scenario 6 DAB ............................................................................................ 89
Figure 75 - Living Room aggregated DAB ............................................................................................. 91
Figure 76 - Outdoor Scenario 1 DAB ..................................................................................................... 92
Figure 77 - Outdoor Scenario 2 DAB ..................................................................................................... 93
Figure 78 - Outdoor Scenario 3 DAB ..................................................................................................... 94
Figure 79 – Truncated Final DAB (removed room logic) ........................................................................ 96
Figure 80 - DomoBus System Specification XML of the use case's Family House .............................. 101
1. Introduction

1.1. Motivation

The popularity of home automation is steadily increasing due to trending topics like the IoT. At home, people want comfort, new entertainment methods, more security and to save time by not being concerned about repetitive tasks. People also want to effectively manage their consumptions (e.g. electric, gas and water) in order to reduce expenditure and reduce impact on the environment. Also, home automation brings new benefits for the elderly and disabled people. [1,2] A smart home should offer great flexibility and control to its users, have the capability to evolve and be adaptable to their needs. It should possess the necessary means of computation, automation and communication to, in an integrated and coherent way, manage the house’s available resources efficiently, empower productivity, save energy and offer the highest comfort and security standards. [3]

There are established solutions that implement robust physical and data-link layer protocols for swarms of edge devices (e.g. Z-Wave, ZigBee, CEBus). However, the existence of multiple alternatives that are incompatible have hindered the standardization process of communication protocols for home automation edge devices. To try to overcome this problem, several movements to attempt normalization have occurred in the past two decades. Multiple manufacturers joined alliances such as ZigBee and KNX and launched products whose specifications were open to interoperability [4]. Nevertheless, these normalization movements still led to concurrent technologies and to lack of out-of-the-box interoperation outside the alliances. Gateways between alliances and their standards are still necessary for their integration and some interactions are hard to develop. For example, Z-Wave and ZigBee are both low power wireless protocols that use a mesh network for extended range, but have difficulty when trying to communicate with different manufacturers, including with one another [5]. Insteon is also a home automation protocol that only specializes on bridging the gap between powerline and wireless protocols [5]. Concurrently, KNX is another large player that entirely defines its own communication media for twisted pair, powerline, radio, and ethernet [6]. The need for interoperable systems becomes evident because many of these will become obsolete and will not allow integration with future dominant technologies, forcing a total change of system upon users [7].

With the popularization of the IoT concept comes an outbreak of new devices from which household items are a potentially huge market. It is expectable that themed packs, with different purposes, will carry dozens of different, low-cost, low-bandwidth, sensors and actuators to be sold per house. Hundreds of different household devices (e.g. kitchen and living room related) are already being embedded with intelligence to become connected. This means that a typical home will have hundreds of devices that will be instantiated on the automation system. Homes will be super-automated in the sense that a large amount of complex behavior and functionality will need to be programmed in the system due to the large number of connected devices [2]. Therefore, home automation systems need to streamline the programming of a great volume of automation rules. They should further provide
adequate tools for that purpose, whereby the end user is the one formulating the house behavior without any programming skills or knowledge about architecture specificities.

Regarding the architecture of local processing solutions (i.e. non-cloud solutions), another large limitation they impose is re-programmability of automations. For example, it is possible to configure a wireless home networking system by using ZigBee alone. According to the work referred in [8], the authors describe a two-tiered approach. The first tier is composed by edge devices (i.e. sensors and actuators) with 802.15.4 ZigBee nodes, 802.15.4 ZigBee PAN coordinators – to route information from the edge devices to Network Controllers – and Network Controllers which are gateways connected to a wired backbone that translate information given by the user into commands for the PAN Controller (and vice versa). The second tier is simple, being composed by the wired backbone and one PC. The PC reads information from the devices for monitoring purposes and from user input to actuate over devices or set configurations. However, the various functionalities and automations are hardcoded into the PAN Controllers. In this case, real-time requirements of these functionalities are guaranteed by the ZigBee modules, but the user can only parameterize the available automations. Therefore, new system behaviors are not possible to be created by the user himself.

The lack of a dedicated rules engine on the architecture, that is reprogrammable and compatible with all possible edge device functionality, imposes a major limitation on the majority of local processing implementations because the user lacks ad-hoc control of the house behavior. There are solutions, nonetheless, such as KNX, that provide not only the standards and communication protocols but also an application for creating automation. Still, this automation creation requires the user to hire professional services for advanced functionality (i.e. complex automation), otherwise the user will only be able to configure predefined rules and specific parameters of each edge device product (whose manufacturer is a member of the alliance).

Recently, cloud based products emerged with the promise to create smart homes by connecting edge devices directly to the Internet, e.g. Muzzley, IFTT, SmartThings and Thington. These are usually composed by a mobile application that serves as a selfcare channel to monitor devices and execute simple actions, and a datacenter infrastructure to process application logic. There are no intermediary control networks, no gateways and no local processing, only edge devices that connect directly to the home router and speak with cloud APIs. Again, even though they offer a rules engine that is reprogrammable, only simple automation chains (if then else alike), such as connecting a light switch to a lightbulb or the garage door to the smartphone’s location, are possible. Their strategy is usually to augment this simplistic approach with parallel services such as machine learning that advises evident parameterizations and addition of simpler automation rules. From the super-automated home perspective, these are not feasible means of automating potentially hundreds of devices with better intelligence than simple cause and effect. These applications are only feasible for a manageable number of devices with very simple activities that can be operated through a smartphone. In reality, a lot of complex and intelligent behavior should happen in the background of smart homes. Additionally, the user cannot modify automations with its own custom logic/functionality but rather only connect predefined functions of one device to compatible ones on another.
In contrast, the DomoBus home automation system is a local processing solution that was used as basis for this work. It is a platform for implementing user created automations on the top layer of its two-tiered architecture – the Supervision layer – which communicates with edge devices in the bottom layer through the Control layer, while providing interoperability between different manufacturers. This work introduces the concept of applying complex automation to the DomoBus home automation system. The DHM application was created to conceive and operationalize a DomoBus system and its intelligence, in which it is possible to import the physical constitution of the house or building, attribute behaviors to rooms, then identify what are the available edge devices and instantiate them on the final house automation.

1.2. Objectives

This work further contributes to the evolution of the DomoBus system and its underlying concepts, having the following objectives:

- Create a software program (DomoBus House Manager) where:
  - end users deploy complex automations while being abstracted of underlying behavior;
  - technical users can create automations from scratch;
  - users can connect any device (from any manufacturer) to the same automations while being abstracted of heterogeneous control network architectures and protocols;

- Create a strategy for deploying automations in a distributed DomoBus supervision level.

1.3. Document Structure

Important home automation concepts are presented in sections 2.1.1 and 2.1.2. In order for the reader to understand what type of standards/protocols the DomoBus control network may interoperate with, a summarized presentation of the most influential home automation standards is given on section 2.1.3 and home automation cloud solutions are presented in section 2.2. In section 2.1.4 the concept of super automated homes is presented. Features for creating automation are expected to be present in smart home solutions, but the way those features are implemented entirely depend on the underlying system architecture. On section 3, the project in which this work is based on is explained in order to equip the reader with the understanding of why certain low-level decisions were made on the work object of this thesis – the DHM – and of its precise position and purpose on the DomoBus big picture. The DomoBus project is currently constituted by various software modules that are being developed and tested separately. However, the DHM will generate code that needs to have an integrated understanding of all modules and layers (i.e. Supervision and Control). The DHM is described in chapter 4 and its analysis is done in section 5.
2. State of the Art

2.1. Home Automation Concepts

2.1.1. Home Automation Systems Overview

A home automation solution is typically composed of small dedicated devices (i.e. embedded systems containing sensors, actuators or control points like switches), which have a series of electrical and/or mechanical capabilities to input signals into the system or exert some action over physical properties, and complex devices (e.g. smartphones, tablets, TVs, coffee machines, refrigerators, etc.) with specific capabilities that are typically mapped to application variables. Consequently, some embedded entity needs to interact with these capabilities either via I/O electronics (if they represent a variable in the physical medium) or directly to application context variables (e.g. application running on a complex device such as a smartphone) to apply new states/values, intermediate processing and/or retrieve data. For example, in the case of small dedicated device capabilities, e.g. the distance measured by an ultrasonic sensor, these need to be measured (in this case by an ADC), interpreted and processed by a microcontroller. Each device also needs to have some type of embedded network interface in order to send and receive data via an arbitrary communication protocol. The entity that interacts with the capabilities typically runs an application that handles network transactions implementing the chosen communication protocol stack.

![Figure 1 – Capabilities and possible interconnections between embedded systems](image)

In a home automation system, one of the main concerns is to have capabilities available to be controlled directly by the user or by a global intelligence program or perhaps by other external applications. There are several approaches to exposing device capabilities that are intrinsically related with architectural decisions, such as the role of embedded systems’ applications, the existence of a complex global intelligence (i.e. a ruleset engine), the location of that intelligence and the network topology.
Table 1 - Architectural decisions for exposing capabilities. With or without Global Intelligence vs Intelligence implemented on Embedded System Layer or not.

<table>
<thead>
<tr>
<th>Embedded systems’ applications don’t implement system intelligence</th>
<th>System has a complex global intelligence concept.</th>
<th>System doesn’t have a complex global intelligence concept.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Client / Multiple Server solution where multiple embedded systems are serving data from capabilities to a single specialized layer where a client — reprogrammable or static – ruleset engine has complex global logic to set or request data to/from servers. This is the type of solution implemented in this work, that has the complete house state at the specialized layer.</td>
<td>Each embedded system can only be actionable/read directly or through simplistic cause-effect rules and/or parameterizable.</td>
<td></td>
</tr>
</tbody>
</table>

| Embedded systems’ applications implement system intelligence | Peer-to-peer type of solution. This is usually implemented on local solutions that have real-time requirements (i.e. where all embedded systems interact directly). This type of solution is complex to implement. Even though there are use cases in literature [9], it’s not discussed in this work since this is not the approached architecture. | Peer-to-peer type of solution used in the first implementations of home automation. System intelligence is hardcoded into functionalities in the embedded systems. The system typically has a very specific scope and it is aimed for direct control of functionalities provided by embedded systems. There is no ruleset engine. |

Initial home automation systems only had embedded devices communicating with other devices of similar/compatible functions, where a local network would form called a control network. They would also implement application logic directly on these embedded devices, not adhering to application logic layered delegation. Nowadays, according to modern reference architectures, local control networks can
still form but have Internet connectivity as well. Communication between house embedded devices with remote servers – that implement storage and application logic – had become feasible with the popularization of technologies like RESTful Web Services, Web Sockets, DDNS, UPnP, even though it raises pertinent security and latency (e.g. for real time applications) issues inherent to TCP/IP connectivity outside a LAN or WPAN and related with datacenter latencies and lack of privacy. Today, the network layer is implemented by the client’s ISP, while the Service & Application support and the Application layers are implemented by third party companies that work on providing interoperation between different device vendors and home automation APIs and GUIs for the user to control his home system. The solutions offered by these types of companies are shown in section 2.2.

A modern home automation system, that is compliant with modern IoT reference architectures, can have IP addressable devices directly connected to the internet whose requirements do not include being part of a local network. For those devices that require it, these should be grouped into control networks that execute simple and very well-defined actions, in order to better identify design flaws and prevent a full system breakdown in case of failure of a specific device. There should be a certain degree of centralization, often applied to the interpretation of rules (simple or complex, as it will be discussed in section 2.1.2) and to hold the full system state, in the form a top-level layer. The achieved degree of control over house capabilities depends on how intelligence can be expressed at the top-layer. For that purpose, capabilities (or properties as referred by the DomoBus naming convention) can be announced as inputs or outputs of a certain type to the overall system by declaring the device and capability via some type of configuration file or database. These concepts will be further explored, specifically using the DomoBus model, in section 3.

2.1.2. Rule-Based vs User-Controlled Automation

According to Table 1, we can look at home automation intelligence through two vectors: Intelligence Type and Intelligence Location. The location of intelligence depends on the topology of the system. We differentiate a layered (and centralized) from a standalone (and distributed) approach. The layered approach centralizes system intelligence at a specialized layer where embedded devices must contact this layer because they are not operating cooperatively nor directly with each other. In the standalone approach, embedded devices process intelligence/automation logic (either complex or simple) in a cooperative and distributed manner, directly using the low-level control network and its protocol for that.

The type of intelligence can be split into two subgroups, the first one being complex stateful rule-based automation, where a programming phase takes place before the user is able to use the system and the second one being simple cause and effect user-controlled automation, where the household member is focused in direct interaction with capabilities, explicitly taking a single action expecting that will cascade into smaller ones. A system may include both types of automation, but they are very different between themselves. Rule-based automation should decrease the number of interactions between the user and the system while in user-controlled automation the user spends more time issuing more commands for
finer grain control. Introducing rules definitely represents an additional level of automation complexity but it also offers abstraction possibilities for truly intelligent home scenarios. Both event-based rules, like turning the lights on when someone enters the kitchen, and time-triggered rules, like turning the sprinklers on at 4 PM, need to be implemented and instantiated in these systems. Current rule-based products configure these rules only at compile time, that is, when someone just bought a house and wants to automate it, for example. This is usually done by a hired professional installer from an outsourced company that deploys the code into the embedded devices [10] [11]. This is a disadvantage in the long term when the user wants to modify rules or expand the network by introducing new devices. Since maintenance is required for a simple update or expansion, solutions that can be dynamically configured at runtime by the user would offer a substantial gain in flexibility, manageability and a major decrease in cost of ownership. User-controlled automation usually degenerates into users creating too many simple cause-and-effect threads that it degrades the experience and control of the overall system. This is a common theme in research related to smart homes: the importance of placing people in control but avoiding the paradox observed by Randall [12], where control systems were so complex that people experienced a lack of control [13].

Even though home automation has been available for over three decades, research has shown that users feel that while home automation has potential benefit for massification, it is still not stable enough for the average person. High cost of ownership, inflexibility, poor manageability and difficulty achieving security are the four main barriers to broader adoption [1], usually brought by functionality being hardwired into the system. Even for the most complex systems available, that include a controller and an interface allowing the user to set parameterizations like schedules for timed events, the offered capabilities are limited and users should have a higher amount of control over their homes in a straightforward fashion [8], being able to define and customize system behaviour, both on rule-based and user-controlled approaches for automation. Most users need to choose between integration ease and flexibility and often structural changes are needed to install home automation. Many iterations are needed before the system setup fulfils users’ expectations and some solutions’ reliability do not portrait the best implementation standards, resulting in unpredictable behaviour and lack of responsiveness, especially in rule-based automation [1].

2.1.3. Home Automation Standards

As shown in Table 1, there are multiple locations for application logic and multiple ways to expose capabilities to it or directly to the end user. In this work we are interested in understanding how home automation standards heterogeneously specify device models, OSI layers and other specifications that allow the application layer to control and automate their devices’ capabilities. An overview of three different home automation standards is shown next, ordered by the time they were conceived.
2.1.3.1. X10

The X10 communication protocol transmits data over low voltage power lines and at a very low transfer rate (around 20 bits/s after allowing for retransmission, line control, etc.), making data transmission so slow that X10 is confined to turning devices on and off or other very simple operations. Data is encoded onto a 120 kHz carrier which is transmitted as bursts during zero crossings of the 50 or 60 Hz AC alternating current waveform. It can contain up to 256 devices and these are addressed by a 4 bits letter and a 4 bits number. Transmitted data consists of this address and a 4-bit command. Commands can only be as simple as turn on light C2 and turn off D5. There are three types of X10 devices, the ones that can only send orders known as Controllers, the ones that can only receive orders known as Receptors (that actuate the electrical device that is connected to it), and the ones that are a mix of both being able to send and receive orders (i.e. receptors with an acknowledgment function). Receptors possess two rotating switches, one with 16 letters and the other with 16 numbers, identifying one of 256 possible addresses. To add a new device, it’s only needed to use the receptors switches and plug in the new device to the power line. In the same network there may be various receptors with the same address, as they will all execute their designated function. [7, 14, 15]

According to the above information, we can conclude that:

- Very low transfer rate limits device capability types and turns real time requirements unfeasible;
- No global system intelligence can exist in a standalone X10 system since the standard does not allow for custom applications at its devices;
- It is not a modular system. Integration of X10 with other standards or with an intelligence provider layer is hindered by the strict definition of 3 device types;
- Data encoding provides few possible commands;
- Simple actuation over electrical devices also reduces capability type choices.

2.1.3.2. KNX

The KNX communication protocol is available for twisted pair, power line, radio frequency, fiber optics and IP/Ethernet physical media. It is a decentralized standard since all devices can communicate as receptors and emitters, without hierarchy and without top-level modules implementing a centralized intelligence layer. Devices communicate with each other directly through packets that follow a format specified by the KNX protocol. Each device possesses its own microprocessor and implementation of the communication protocol for media access (CSMA/CA policy). Addressing a device can be resumed into Area (from 1 to 15), Line (from 1 to 15 where) and Device (from 1 to 255). There are also Group Addresses, where the same Group Address can be given to any number of actuators and they will all respond to a single order. KNX has its application logic distributed throughout all devices’ microprocessors since the house behavior is divided and committed to each specific device at the moment of installation. It has a dedicated programming tool, the ETS, where a specialized installer can be hired to reconfigure interactions between devices according to the capabilities of each one. The ETS uses detailed information of each device (can be imported from the hardware supplier), binds an
individual address or a group address to each one and it allows to individually parameterize them according to the manufacturer’s datasheet, as well as to import programs from the suppliers for device configuration. [10, 15, 16]

According to the above information, we can conclude that:

- Wide range of available capability types because KNX compatible devices always have an embedded system running the KNX application stack, enabling the parameterization and configuration of read/write operations over these capabilities via ETS;
- The wide range of available communication media offers multiple transfer rates and latencies, making real time requirements feasible for certain configurations;
- Integration of KNX with other standards is possible since a gateway device can be introduced to the system as a KNX compatible device with its own running application to translate communications;
- Implementing complex stateful interactions between devices (i.e. maintaining a refined global intelligence) in a standalone KNX system is unfeasible since KNX is tailored for simplistic control networks based on action-response. ETS is tailored to only allow parameterization of application capabilities, directly design which switches/sensors activate certain actuator addresses and to receive feedback of capability statuses (after actuation);
- Integration of KNX with an intelligence provider layer can be achieved by implementing a gateway node that translates between the two layers and by manually programming the KNX subsystem via ETS. Actuation would only be based on commands coming from the gateway and all sensed data would only be sent to the gateway (i.e. stripping all device interaction logic from the KNX subsystem and delegating it to the upper layer).

2.1.3.3. ZigBee

Zigbee is a low-power, low-data rate, low-latency and close proximity wireless ad-hoc network standard, targeted at battery-powered embedded devices in wireless control and monitoring applications. Data rates vary from 20 kb/s (868 MHz band) to 250 kb/s (2.4 GHz band), being best suited for intermittent data transmissions from sensor/input devices and applications that require long battery life and secure networking (network packets are encrypted with AES 128-bit) [17]. Zigbee builds upon the physical layer and media access control defined in IEEE 802.15.4 adding four additional components: network layer, application layer, ZDO and manufacturer-defined application objects [18][17]. There are three kinds of Zigbee devices: ZC, ZR and ZED. ZEDs contain just enough functionality to talk to the parent node (ZR or ZC) and cannot relay data from other devices, therefore being asleep most of the time, having long battery life, requiring the least amount of memory and being cheap to manufacture. ZRs act as relays and run application functions. ZCs are the root of a network tree and may bridge to other networks, existing only one ZC in each network, storing information about it and acting as the repository of all security keys. The network layer enables the correct use of the MAC sublayer and provides an interface for use by the application layer. Its capabilities and structure are those typically associated to network layers, including routing, handling configuration of new devices, establishment of new networks.
(i.e. tree, star or mesh), determining whether a neighbouring device belongs to the network, discovering new neighbours and routers, etc [17]. The average star topology (with packet size 32 bytes) has an average latency of 58.5 ms whereas the round trip delay is 92 ms [19]. The Zigbee application layer is the interface between the Zigbee system and its end users, comprising the ZDO, which defines the role of a device and its offered services, and manufacturer-defined application objects. This layer binds tables, sends messages between devices, manages group addresses, reassembles packets and transports data, while also being responsible to provide service to Zigbee device profiles. For applications to communicate in order to achieve a larger goal, their comprising devices must use a common communication protocol. These sets of conventions are grouped in the previously mentioned device profiles [17, 18]. There is also an APS, that sits above the network layer, which understands applications. The APS frame includes endpoints, profile IDs, groups, etc. The APS is responsible for filtering out packets for non-registered endpoints or profiles that don't match, generating end-to-end acknowledgment with retries, and other relevant features [20].

According to the above information, we can conclude that:

- The application profile framework allows different vendors to independently build and sell ZigBee devices that can interoperate with each other within a given Zigbee application profile [17]. On the other hand, while integration with other standards is possible, it can be faulty or hard to accomplish. As shown in [21], a KNX-Zigbee gateway node served as both KNX and Zigbee translator and was incapable of some attribute and services translation;
- Features such as application profiles and the ZDO demonstrate that a home automation system using Zigbee devices has a wide range of capability types available since these are fully configurable in each Zigbee application;
- Zigbee’s wireless communication media average latencies and round trip delay times depend on the topology and packet size, compromising real time requirements for certain configurations;
- Implementing complex stateful interactions between devices (i.e. maintaining a refined global intelligence) in a standalone Zigbee system is possible but unpractical. A Zigbee embedded node has low memory, slow CPU speed and low data rate. The application code of a centralized rules engine would run and communicate slowly for super-automated environments (i.e. very large number of devices and interactions). Zigbee is tailored for operating as a control network in low power consumption mode with intermittent data transfer and the APOs are made to directly control hardware, not handling complex automation.
- Integration of a Zigbee system with an intelligence provider layer is feasible since the gateway node would be the ZC with a specialized application for translation between the two layers (e.g. similarly to KNX-Zigbee gateway on [21]). Integration strategy’s the same as mentioned on KNX.

As we can conclude, the home automation standard used in the control network of devices and its inherent architecture heavily influences the feasibility and effort for implementing global intelligence – based on a static or re-programmable ruleset – in standalone or layered patterns. In other words, in many cases it is not feasible, or is extremely hard, to implement global intelligence in a standardized control system for embedded devices without additional specialized nodes operating externally.
Since the object of this work introduces a complex global intelligence model that is not distributed throughout the embedded systems, from now on we will narrow our discussion to Single Client / Multiple Server solutions that have a single specialized layer that manages the house state, ruleset engine and introduces a common communication protocol. Each device – in its own control network – is serving capabilities to a ruleset engine on the previously mentioned layer that sets or requests data to/from these servers. This intelligence provider layer is then the only translation point between all device control subsystems/standards, e.g. in case there are multiple home automation protocols such as KNX and Zigbee. This solves the problem of global intelligence placement and the problem of interoperability between different home automation standards by reducing the multiple number of bridges between different control network standards to only one flexible protocol in the intelligence provider layer. The job of control networks gets simpler because in this scenario there should be mostly simple actuation commands being sent from the ruleset engine to the control network and sensor data sent from the control network to the intelligence layer. Note that control networks can still implement implicit intermediate logic between their devices and present the result as a function for automation on the intelligence layer. An example of this would be the distributed lighting control as shown in [22], where a local PID (Proportional-Integral-Derivative) controller computes a dimming value on each luminaire device – composed by an LED and a photo resistive sensor - and a global control algorithm is computed in a distributed fashion on every device of that control network.

2.1.4. Super Automated Homes

As previously discussed in 1.1, these challenges span from the interconnection of multiple heterogeneous networks composed of many (physical) objects (e.g. sensors, actuators, microprocessors, SoC, personal computers, servers, etc.) to application logic challenges inherent to facilitating automation of the system behavior using all possible interactions. The automation problem gets specifically difficult for home automation solutions since homes can have many control points and devices, even for medium sized ones.
This rather feasible example of a basic living room IoT package accounts for a sum of 42 addressable nodes, each with its specific set of automation enabled capabilities. Considering that a house has simpler divisions (with approximately half of these nodes), such as bathrooms and hallways, but that the kitchen and bedrooms are also complex, we can estimate that for a T3 type of home (with 3 bedrooms, 1 kitchen, 1 living room, 2 bathrooms and 1 hallway) there will be more than 273 addressable nodes. This number will greatly increase for detached houses, which will probably have more inside area and a garden [23]. Also, each node has several automatable attributes/parameters (in the hypothetical scenario where there is an average of 2 generic parameters per node, e.g. on/off and intensity, there will be 546 automatable attributes/parameters). The manageability of automations by the end user decreases with this number. Even though this number is higher when compared to common basic automation solutions, we want to address a much bigger number and offer more abstracted automation levels [23]. Given our aim, we introduce the concept of "super-automated" homes that applies to homes with an extensive and rich set of automatable parameters:

Super-automated homes are composed of a very high number of automation rules, with complex interconnections. The large number of rules is naturally related with the extensive amount of edge devices and their exposed capabilities.

This spawns a set of requirements – which this work aims to accomplish – for creating and managing automation on these homes, such as having a behavior builder program that eases the construction of complex interconnected logic and makes it reusable. Also, it should abstract developers and end users from the integration details of different device manufacturers and facilitate the continuous creation and management of rules and their relationships, among multiple other requirements that were presented on section 1.2.
2.2. Cloud-Based Home Automation Solutions

2.2.1. Muzzley

Muzzley is a Portuguese startup that provides a cloud service to connect home devices. It offers a desktop and mobile application front-end to remotely control devices, monitor them and create simple automations for the smart home. Muzzley does not sell any edge devices of its own neither it enforces any specific communication protocol upon its associated manufacturers. This happens because Muzzley does not require any hardware for its infrastructure on the user’s house since the edge devices connect directly to the home router LAN (except when the vendor has its own hub). At its datacenters – broadly referred to as cloud – is where all the application logic lies and where all the devices’ interfaces are defined according to Muzzley’s device hierarchy. This segregation of the application logic is typical of cloud services. Therefore, Muzzley’s strategy is purely based on integrations. Muzzley has them with many companies like Nest, Philips, Insteon and FitBit, so its core business is to maintain its service easily open for integrations with vendors’ devices or clouds. Hence the vendors must embed their devices with microprocessors, network cards and message interpretation logic for them to be connected and for their capabilities to be exposed. Like previously mentioned, whether devices need hub support or not is independent from Muzzley. Some products do need ZigBee or Z-Wave hubs in order to work. Moreover, Muzzley claims its differentiation lies on applying machine learning algorithms to automations:

We use machine learning to adapt to your lifestyle. Based on your surroundings and context (e.g. location, time of day, weather), as well as the devices you add in Muzzley, we’re able to suggest customized automation recommendations. This will help you manage your home more efficiently [25].

These recommendations are called Intelligent Cards which suggest useful but rather simple automations between devices. For example, an automation card can suggest connecting the current mobile phone location to the thermostat, so it starts heating the home before the user arrives [24]. An intelligent card is essentially a recommendation of an agent implementation. A Muzzley agent is an automation between devices – a set of simple conditions over some devices that, if verified to be true, trigger a set of actions on other devices.

Figure 3 – Defining a Muzzley agent on its mobile application UI [25]
Manufacturers who want to partner (i.e. integrate) with Muzzley must undergo device/cloud registration which will be summarized below with the degree of detail of the available online documentation.

There are two processes for integrating a new device type in the Muzzley ecosystem: Cloud-to-Cloud for IoT device manufacturers, which typically have a complete infrastructure already built to support communication with their devices, and Cloud-to-Device for independent developers looking to control and automate their devices without having to build a complete system with mobile applications. [25]

In both types of integration, there are two components the manufacturer always needs to develop: [25]

- **Manager** – the program where all manufacturer-specific logic will reside, acting as the gateway with Muzzley’s logic. It needs to have an HTTP interface, that will make and receive requests from/to the Muzzley’s HTTP REST API, and a Realtime Communication Interface using MQTT, so that Muzzley’s users can interact with the manufacturer’s IoT devices;

- **Device specifications** – these are setup at the Selfcare site after registering an account. All device specifications, such as what type of device it is, how to interact with it and what interface it uses, are set up here so that Muzzley’s cloud (i.e. other manufacturers) can interact with it.

Before providing details about the nature of the manager and device specifications, it is best to understand Muzzley’s Device Hierarchy:

*Figure 4 - Connections between Muzzley’s Cloud (selfcare channel), the manufacturer’s manager (gateway) and the manufacturer's cloud or device [25]*

*Figure 5 - Muzzley's device hierarchy, the breakdown of a device [25]*
- **Profile**: the unique identifier of the device type (i.e. the manager’s identifier);
- **Channel**: the unique identifier that allows communicating with a concrete device;
- **Component**: the identifier for a certain component of a given channel (e.g. fan, thermostat, …);
- **Property**: the identifier for a certain property of a given component (e.g. angle, speed, …).

Regarding the registration of device type specifications on the selfcare site, it can be divided into: [25]

1. Details of the integration – where generic information about the device type is specified:
   a. **Integration type**: Cloud-to-Cloud or Cloud-to-Device;
   b. **HTTP URLs**: Endpoints to communicate and get information about the device/cloud;
   c. **Interface UUID**: The identifier of the selected interface to be used when users interact with the device via Muzzley’s mobile application;
   d. **Required capability**: how devices are added to Muzzley (through a web interface with username/password or through UPnP lookup in the user's local network). This allows the device to only be shown in Muzzley clients which support the selected capability;
   e. **Email access list**: whitelist to indicate which Muzzley users should be able to view the device while it is not publicly available;
   f. **Auth**: contains security data that allows communication with our HTTP API.

2. Ontological specifications – establishes relations between the different components and properties of the device type, so that Muzzley can interpret its properties and actions (see annexes for a detailed view);

3. Used interface – Muzzley’s implementations are independent from the interface the manufacturers create. Therefore, an editor is provided after creating an interface in the selfcare page (see annexes). There are JavaScript objects that allow user interaction with a device:
   a. Muzzley (global)
      - `muzzley.ready(handler)` – Executed when the web view is ready. The handler receives the `options` object provided by the Muzzley mobile applications;
      - `muzzley.subscribe(options[,callback])` – Allows subscription of a channel. Returns a communication `Channel` type object,
      - `muzzley.publish(message[,callback])` – Allows publishing data to a channel;
   b. Channel
      - `channel.on(event, handler)` – Enables handler registration of specific events;
      - `channel.off(event, handler)` – Disables registration of a specific event listener;
      - `channel.unsubscribe(callback)` – Allows unsubscribing to published events;
   c. PubSubMessage
      - `pubSubMessage.getNamespace()` – Gets the namespace of a message;
      - `pubSubMessage.getPayload()` – Gets the JSON payload of the message. The message is received on the `event` object, handled by the `handler` function.
The manufacturer’s initial setup, to add a new manager to his account, requires a GET request to be made (using Hawk credentials) from Muzzley to the manager’s HTTP server, e.g. https://my-manager.pt/authorization?user=1, where my-manager.pt/authorization is the manager’s URL and ?user=1 is the query parameter with the id of the manufacturer’s user trying to add the device. The response is the URL to which the user should be redirected to perform the login (this page has only manufacturer’s logic and presentation). If the authorization process was successful and the manager’s answer is https://channels.muzzley.com/authorization?user=1&success=true, then the manager is allowed and able to make requests on behalf of the user. Afterwards, another request from Muzzley to the manager must be made, https://my-manager.pt/channels?user=1, requesting all the available channels since the manager will manage several concrete devices of that specific type. The JSON response must have the exact structure as defined in the device(s)’ ontological specifications at the selfcare channel. The manufacturer can then subscribe/unsubscribe a certain channel by sending a POST request from Muzzley to https://my-manager.pt/subscriptions?user=1 with a certain JSON payload indicating the subscription status. In parallel, IoT devices’ actual control is done with MQTT so the manager must also implement topic publishing and subscription, using Muzzley’s well defined MQTT topic pattern v1/iot/profiles/<profileId>/channels/<channelId>/components/<componentId>/properties/<propertyId> (see annexes), and connecting to the mquitts://geoplatform.muzzley.com endpoint. Each physical device will only communicate with its manager using the manufacturer’s chosen communication protocol. [25]

According to the above processes, the prime value of Muzzley’s solution for controlling and creating simple automations between devices is that it eases the interoperability problem between different vendors or cloud solutions. Manufacturers can launch their products, upgrade them or setup their clouds with a Muzzley integration and they will be able to interact with devices from any other manufacturer that is a Muzzley partner. In essence, the overall system (i.e. Muzzley’s Cloud) is agnostic to different communication protocols and media, and it places the responsibility on the manufacturer for registering each one of its devices in Muzzley and enabling each device with communication capabilities. This is essentially enforcing the configuration of a manufacturer gateway on the Muzzley cloud – a parallelism that can be made with DomoBus gateways on a SM. It is still possible to integrate standards such as ZigBee, whose ZigBee edge devices need a local network infrastructure composed by ZigBee PAN coordinator nodes, ZigBee router nodes and a gateway (or as Muzzley calls it, a hub).

What Muzzley does not guarantee:

- Muzzley’s team internally manages its automation engine that allows users to establish rules via UI (using pre-established triggers like time, location and other devices’ outputs). This feature is not open to developers. They can only configure their device information at the selfcare channel for Muzzley to know how it acts, what it triggers and what are its actions – agents automatically use the automation engine to create simple cause and effect rules.
- Muzzley doesn’t support establishing/creating complex automations between devices in ample scenarios. It rather focuses on controlling/interacting with single devices and managing the house state via UI since there is not a rich editor for the house behavior (i.e. it doesn’t address
the super automated home problem, overwhelming the customer when the number of devices increases in the future).

- Additionally, its advertised differentiation factor – the automation recommendations or intelligent cards – only suggests simple cause and effect rules over devices and therefore do not bring additional value regarding more abstracted and intelligent scenarios.

### 2.2.2. Samsung SmartThings

SmartThings is an American company, acquired by Samsung, that provides a cloud service and a proprietary gateway to connect home devices. They offer a mobile application front-end to remotely control devices, monitor them and add simple automations to them. The gateway hardware connects directly to the home’s internet router, speaks with the SmartThings cloud via HTTPS and is compatible with communication protocols such as ZigBee and Z-Wave, allowing sensors and actuators of these other home automation standards to communicate with the SmartThings cloud service. Similarly to Muzzley, SmartThings does not produce any edge devices of their own, but they advertise edge devices from certified/compatible manufacturers belonging to their catalog. They also do not enforce any specific communication protocol upon its associated device manufacturers. The device-to-cloud connection approach is done purely via HTTPS using REST API calls – they don’t use MQTT like Muzzley – to either the SmartThings gateway (for hub-connected devices) or through their manufacturer’s cloud (for IP addressable, cloud-connected devices). All application logic and devices’ interface definitions are located remotely at the SmartThings cloud environment. Their strategy is thus also based on maintaining its service open for integrations with vendors’ devices or clouds – like Muzzley – with the addition of a Samsung certification program called “Works With SmartThings” to curate their catalog of available device models. Vendors focus on producing embedded systems with the necessary networking and message interpretation logic to connect them and for their capabilities to be exposed and controlled.

The SmartThings ecosystem includes the following main components: [26]

- **Devices** – these connect to the SmartThings cloud, either Zigbee or Z-Wave devices or cloud-connected devices. A SmartThings cloud connector integrates devices through their own cloud.

- **Automations** – these can be a WebHook or an AWS Lambda function that uses the SmartThings REST API to control and get status notifications from device, allowing the user to control their SmartThings system without manual intervention.

- **SmartThings app** – centrally manages the SmartThings catalog of devices and automations. It can be used to configure automations and directly control devices.

- **SmartThings API** – consists of REST APIs that enable integration, control and monitoring of devices and services on the SmartThings Cloud.
Different kinds of devices can be integrated with SmartThings by a developer or manufacturer: [26]

- If the device is already connected to a third-party cloud then it’s called a cloud-connected device. The basic process to integrate a cloud-connected device with SmartThings Cloud starts with creating a device profile on the Developer Workspace. A device profile contains the components, capabilities, and metadata that define a SmartThings device:
  - Metadata:
    - Device Profile Name – used in the app, must be identifiable by the end user.
    - Device Type – determines the device’s icon and default UI layout in the app.
    - Vendor ID – alphanumeric identifier for the device.
  - Components – a list of components that will have capabilities associated to them.
  - Capabilities:
    - Commands – represent ways to actuate the device.
    - Attributes – represent state information or properties of the device.
- If the device operates over ZigBee or Z-Wave then it is called a Hub-connected device via a SmartThings hub. At the time of this writing, there are no available developer tools or guides for independent creation and testing of hub-connected devices. This means that the manufacturer needs to work in conjunction with SmartThings for the development of the Device Plugin and Device Profile, for the upgrade of the SmartThings hub code that translates ZigBee or Z-Wave to HTTP REST calls for that new device, as well as for testing and publishing of that new device on the catalog. Only cloud-connected devices will be analyzed further.

Capabilities are created and maintained by the SmartThings development team and should be added to the device profile. An example of a Fan Speed capability definition by SmartThings is shown in Figure 54 (see annexes). We can see that this capability has one attribute called “fanSpeed” of type NUMBER and that it is required; has one command called “setFanSpeed” that it is required and in which the resulting deserialized function receives an argument called “speed”. The data types associated with capabilities’ attributes and commands are also defined by SmartThings and they are currently STRING, NUMBER, VECTOR3, ENUM, DYNAMIC_ENUM (i.e. a set of ENUM that can change overtime),
COLOR_MAP (e.g. [hue: 50, staturation:60]), JSON_OBJECT and DATE (usually represented as java.util.Date object). When assigning capabilities to a device profile, the developer can assign a capability to the Dashboard State and Dashboard Action which will generate a default UI for that device in the mobile app. [26]

Once the device profile is created, a connector must be created using a SmartApp. That SmartApp will be used to integrate the third-party cloud-connected devices with the SmartThings cloud and can either be an AWS Lambda function or a WebHook endpoint with a RESTful API interface (i.e. an API endpoint on the internet which can receive incoming HTTPS POST requests). The application / API endpoint can be written and hosted using the programming language and tools of the developer’s choice. The SmartApp is essentially an implementation of lifecycle event handlers and handlers for events coming from the device’s cloud. A SmartApp comes to exist when it is successfully registered/published in SmartThings Cloud. The SmartThings Cloud will issue a POST request to the SmartApp during various lifecycle phases. The request body will contain the lifecycle that triggered the execution, along with other data depending on the specific lifecycle phase. [26]

The lifecycle phases are the following: [26]

- **PING** – occurs when the SmartApp is registered for verifying its existence and integrity. SmartApps must provide a JSON response with the challenge code that was sent to them.
- **CONFIGURATION** – occurs when the end user chooses to install the SmartApp (i.e. instantiates an Automation or New Device). It will provide the user with basic information about the application, as well as request any information or access to devices the application may request. It has two phases:
  - **INITIALIZE** – occurs at the beginning of the CONFIGURATION lifecycle, as the user begins the installation and configuration process. The SmartApp must respond with JSON specifying basic information about the app, including the permissions required to configure it, and the ID of the first configuration page.
  - **PAGE** – All information that the user may configure for the SmartApp is organized in pages. Occurs for each defined page. The SmartApp is responsible for providing a JSON response with the configuration data for the requested page.
- **INSTALL** – occurs when the end user has successfully installed the SmartApp (i.e. instantiates an Automation or New Device). It is during this lifecycle that subscriptions, schedules or other initialization activities should occur. The INSTALL lifecycle phase request contains information about the installed SmartApp, including all configuration selections and authorized permissions. The SmartApp should provide a JSON response with a 200 OK and an empty installData object.
- **UPDATE** – occurs when the end user updates an already installed SmartApp’s configuration. Has the same handling as INSTALL.
- **EVENT** – occurs when the SmartApp is executed in response to a subscribed-to-device event or scheduled execution. The EVENT lifecycle phase request contains information about the installed SmartApp, including all configuration selections and authorized permissions, as well
as information about the event that triggered the execution. The SmartApp’s JSON response should be a 200 OK with an empty eventData object.

- UNINSTALL – occurs when the SmartApp is uninstalled by the end user. Any cleanup tasks that the SmartApp requires when a user uninstalls the application should be handled during this phase. All subscriptions and schedules for installed app will be automatically deleted. The SmartApp’s JSON response should be a 200 OK with an empty uninstallData object.

A device state can change because of an Automation, an end user interaction with the mobile app or an end user manual actuation / third-party actuation (e.g. external application).

---

**Figure 7 - Issuing commands to a device via Mobile Application or Automation on SmartThings [26]**

As seen in Figure 7, for mobile application or Automation generated commands, the command is first sent to the SmartThings API, which then calls the SmartApp Connector. The Connector then relays it to the third-party cloud, actuating the device and then reflecting that on the user’s mobile application. [26]

However, in the case of manual or third-party device actuation, this sequence will be initiated by the third-party cloud, informing the Connector that the device state has changed – which explicitly handles events corresponding to this third-party-initiated state change or polls the third-party cloud at regular intervals. Upon being notified (by the device cloud or by polling) that the device state has changed, the Connector must issue a POST request into SmartThings API to create the corresponding event. [26]

---

**Figure 8 - Device state change initiated by third-party cloud or mobile application / Automation (modified) [26]**
SmartApps can also work as Automations. Automations allow a user to control their SmartThings system without manual intervention by using the SmartThings REST API to control and get status notifications from devices. The implementation of an Automation is versatile, but usually involves issuing commands to devices’ capabilities using following process: [26]

1. **Set command permissions** – when a user installs the SmartApp, they will select a device to control. The OAuth scope is specified in the INITIALIZE sub-phase lifecycle event request;
2. **Get OAuth token** – the JSON object “authToken” is specified in the INSTALL event request. This string is required in the command HTTPS request header;
3. **Obtain ID of the device** – use the “deviceId” JSON object specified in the INSTALL event request in order to construct the URL request for the device command;
4. **Construct POST request URL** – https://api.smartthings.com/v1/devices/{deviceId}/commands
5. **Construct POST request body** – using the information in the capabilities reference, e.g. for a Switch capability there are two commands without arguments “on” and “off” (see annexes);

According to the above, SmartThings’ prime value is – much like Muzzley’s solution – interoperability between different vendors or cloud solutions. Manufacturers can launch their products and later setup cloud-side logic on their side to integrate with the SmartThings system so their devices will be able to interact with devices from any other manufacturer that are also in the system. Like Muzzley, the system is agnostic to different communication protocols and media, placing the responsibility on the manufacturer for registering each one of its devices and enabling each device with communication capabilities. In terms of control and automation between devices, the features remain restricted to either directly controlling an added device or assigning a group of sensor devices to a SmartApp Automation, setup conditions and choose a group of actuator devices and intended actions on them.

What Samsung SmartThings does not guarantee:

- The SmartThings Application only allows to setup parallel actuation based on the same set of conditions on a SmartThings Automation. Inputs to Automations can only be from sensor devices or schedules so it does not allow for Automations to serially invoke other Automations.
- The solution does not scale well for super automated homes. The customer will get overwhelmed when the number of devices increases, attempting to control hundreds of singular devices directly or attributing standalone automations with no interconnection between them (in such a way that would abstract automation management from the customer).
- SmartThings does not offer a rich editor for the house behavior, so complex automations between devices in ample scenarios cannot be established/created. This further aggravates the super automated home problem because the product only focuses on controlling/interacting with single devices and managing the house state via UI.
2.2.3. IFTTT

IFTTT (If This Then That) is a web service aimed for generic integration of applications with a REST API exposed to the Internet. Not being built specifically for IoT devices – even though it become widely used for that purpose – IFTTT has many available integrations with services such as Gmail, Facebook, Instagram and Pinterest. It offers the ability for the end user to create a chain composed of a simple conditional statement and an action, called applet. Users can add applets to the public catalog or keep them private. There are over 600 services in IFTTT available for integration in an applet. Developers can create new services and if these get approved and published, they can later gain insights into how consumers use their produced services inside applets. [27]

![IFTTT Applets work with everything](image)

**Figure 9 – Screen flow of a user creating a custom applet on the IFTTT mobile application**

Applets can only connect two services. They are managed and monitored by end users on the IFTTT mobile application and they can be created from scratch or imported from the public catalog. Applets can be monitored and controlled directly on the mobile application as well. Users must authenticate to each service used on an applet, if applicable, in order to use the applet. [27]

As shown in Figure 9, the user can create a custom applet on the mobile application. The first and second screens are for choosing one source service, the third screen is for adding a trigger and the fourth screen is for determining the trigger field value. After completing this sequence related with the source service, the flow shows this sequence of screens again but this time for choosing the action.

Applets are composed by: [27]

- **Title** – tells the user what they should expect the Applet does for them;
- **Description** – adds details about triggers and actions, as well as other information the user should provide access to such as location;
- **Trigger** – select a trigger event on the source service;
- **Trigger fields** – one or more fields specific to the trigger (service) that will act as conditions;
- **Action** – select the action on the target service that will be triggered when the trigger event occurs;
• Action fields – one or more fields specific to the action (service). These can be pre-filled or filled out by the user.
• Ingredients – one or more attributes emitted from a given trigger. E.g. for the trigger “New photo added to album” there would be ingredients for “PhotoURL” and “TakenAt.”

IFTTT defines a concise protocol for the developer’s service API to implement. A trigger or action (used in applets) will have its own API endpoint on the service, built specifically for IFTTT. Action endpoints will be writable endpoints that IFTTT will send data to. Trigger endpoints will be event streams that IFTTT will poll for new data by default, unless the service implements the Realtime API specification. The Realtime API is used to notify IFTTT (push based) of new available events for a specific user_id or trigger_identity rather than relying on the default pull based approach. Each time there is a new event for IFTTT to consume the service makes a POST request to https://realtime.ifttt.com/v1/notifications and includes the X-Request-ID and IFTTT-Service-Key headers, as well as the JSON-object payload containing an array of objects which have a user_id or trigger_identity. [27]

The workflow for developers to add a service to IFTTT is as follows: [27]

1) Create a partner account in IFTTT.
2) Set up the service’s execution environment – spin up an execution environment of your choice in order to expose IFTTT-specific public API HTTPS endpoints that correspond to each trigger and action in the service and process IFTTT requests.
3) On the partner account, register the new service by filling the API URL prefix and indicate whether the new service implements the Realtime API hook (and retrieve the Service Key in that case). Use IFTTT’s OpenAPI 2.0 definitions with a bootstrapping tool (e.g. Swagger Codegen) to accelerate development of the service.
4) Build the status endpoint (i.e. /status suffix) on the service and then run the Endpoint Test on the IFTTT partner account webpage. A passing status indicates that IFTTT can reach the service’s API.
5) Describe and build the service’s triggers, their ingredients and the actions. If the service requires authentication, then storage of user information must be implemented on the service back-end and an authentication flow must be set up using OAuth2 for IFTTT to make requests to the service on behalf of the user.
6) Test the service either by using the IFTTT endpoint testing tool or by creating Applets using each available trigger and action from the service. There will be a special user available for this purpose with preview access to use the service on IFTTT applets.
7) Submit the service for review.

Regarding the IoT and Home Automation landscape, IoT device vendors will create an IFTTT service which is hosted on their own cloud environment – requiring user authentication on applet installation – therefore allowing IFTTT users to read from or write to devices they bought from the IoT vendor via authenticated IFTTT applets. The vendor’s service must provide basic functionalities, such as matching both accounts (IFTTT and the vendor’s) and use trigger/action fields to identify the device.
What IFTTT does not guarantee:

- The IFTTT Application and applets were tailored for setting very simple chains of conditions on a source service to actuate on another service. Serially interconnecting applet outputs and inputs is not possible, substantially limiting any slightly complex automation.
- This solution does not scale at all for super automated homes. There is no rich editor for the house behavior. The customer will get overwhelmed when the number of devices increases, having to manage hundreds of applets and their one-to-one service mappings, with no automation management/abstraction features.

3. DomoBus

3.1. Overview

DomoBus is a project that specifies a home automation system and provides proprietary software to implement it. It defines the system architecture, software architecture, house model, device model, communication protocol and communication infrastructure requirements. Its intent is to address super-automated homes in an efficient and cost-effective way [2]. DomoBus predicts heterogeneity between infrastructure hardware components, specifically between supervision nodes and control nodes. Therefore, its control layer software is best suited for inexpensive, low consumption and low resource hardware (typically microcontrollers), while its supervision layer software makes use of OS resources. A control process (on a CM) has been deployed on a AT90S8515 microcontroller from ATMEL with 8KB of FLASH program memory, 0.5KB of RAM and 0.5KB of EEPROM [2]. These are outstanding results when compared with today’s available capabilities of microcontrollers, easing the dimensioning problem.

Currently, according to the previously mentioned deployments and with no in-depth analysis, it is safe to say that today’s hardware has enough resources to withstand thousands of edge devices (i.e. sensors and actuators), considering that a CM (Control Module) will not perform complex system-wide operations but will rather report events to the supervision level and execute simple microcontroller-level actions on edge devices [2]. Nevertheless, the recommended dimensioning for the number of properties, expressions, actions, etc. on a supervisor node, as for the number of devices connected to a CM, should be studied in the future once the system is massively scaled.

The development of the DomoBus home automation system aims to mitigate many difficulties felt while deploying and using commercially available solutions. Frequently, specialized development tools or specific integrated circuit boards were needed, which were not readily available and had a steep learning curve, even for the more technically curious users. [23]
The main advantages of DomoBus when compared with current solutions are: [23]

- specialized functions and infrastructure to handle super-automated and real-time environments;
- ability to interconnect with distinct control networks to accomplish interoperation;
- security of exchanged data between edge devices, supervision process and storage/database;
- modular installation of SMs and CMs in each room to simplify cabling/wireless to edge devices;
- distributed supervision level allowing to share load, increase responsiveness and eliminate the single point of failure problem.

With this work, as mentioned on the objectives, DomoBus gains the additional macro advantages:

- end users can deploy complex automations while being abstracted of underlying behavior;
- technical users can create automations from scratch;
- users can connect any device (from any manufacturer) to the same automations while being abstracted of heterogeneous control network architectures and protocols;
- a strategy for deploying automations in a distributed supervision level.

3.2. System Architecture

The DomoBus system architecture is layered and distributed by design. There are two layers with separate functionalities and purposes: Supervision Layer and Control Layer. The Supervision Layer is composed by multiple SMs which communicate in this layer in order to keep the house state distributed by them. The Control Layer is composed by multiple networks of CMs which will communicate with one SM and specific sensors & actuators.

3.2.1. Control Layer

The Control Layer is composed by multiple networks of CMs, each network potentially being a distinct vendor subsystem. A CM is typically hardware with dedicated memory, timers, communication interfaces and an embedded microprocessor, which will connect directly to sensors & actuators such as switches, movement detectors, temperature sensors, servo motors, lights, among others. All CMs inside of a network implement the communication protocol of that vendor’s subsystem (which is also implemented on the DomoBus physical gateway). The CMs inside of a specific network are connected to a dedicated DomoBus physical gateway for intercommunication with the Supervision Layer. Each one of these physical gateway devices is connected to one SM via USB and handle data transfers with the different physical medium of the CM network, i.e. between the Gateway Process on the SMs and the applications running on the CMs. The physical gateway will also adapt the addressing model of the vendor to DomoBus’ device model (as shown in section 3.3).
The DCN was created within the DomoBus project to be used as its native control network – which offers several improvements regarding control networks, such as the ability for CMs to run multiple applications and interface with multiple end devices – and to have a starting point for interoperability testing between different control network vendors. In DCN, each CM executes different applications which are responsible for controlling a variable number of end devices, where each application executes tasks in response to received messages and notify the supervisor of any changes that occurred in the devices. These applications implement simple state machine mechanisms which enable cyclic round-robin scheduling on the non-preemptive microcontrollers of CMs, allowing for the interruption of a certain task without losing information and enabling finer-grain control over real-time requirements. However, due to memory restrictions, it is not possible to perform system-wide actions in response to a given event in these applications, delegating that responsibility to SMs. Nevertheless, another one of DCN's advantages (which is typically not available on CMs of other vendors) is the ability to implement CM applications with some intelligence and abstract away the involved sensors and actuators as a virtual device (and its virtual properties) to the Supervision Layer. As per the example mentioned in 2.1.3, a distributed lighting control application can be installed on CMs so that these implement intermediate logic between their devices and present the result as a function (virtual device-property pair) for automation on the Supervision Layer. [2]

Each vendor's control network will have its own communication protocol and software. This work does not require analysis of specific control network communication implementations – since it focuses on
the Supervision Layer – but rather only knowing the interface details provided by the DomoBus physical gateway, the exchanged message types and their associated events.

In Figure 11, the structure of a device address when interacting with DCN is shown. Every device address is 32 bits and must be globally unique. The first 16 bits represent the supervision-level gateway application that handles the device’s messages and the second group of 16 bits represents the control-level device address which refers to a specific physical gateway (i.e. Node Address), a specific application running on a CM (i.e. Node Application) and a specific virtual device on that Node Application (i.e. Application Device). Other control networks use a different control-level device address structure.

CMs on an arbitrary control-level network will implement different message types and a protocol that the DomoBus physical gateway will adapt to. On DCN, messages are sent from the CMs to the DomoBus physical gateway using the DCN packet format. On other control networks, the DomoBus physical gateway will either be notified of events on a different packet format – which will then send to the Gateway process at the SM for translation to DomoBus supervision-level packet format – or will apply different mechanisms to poll events of CMs and then send to the Gateway process at the SM for translation to DomoBus supervision-level packets.

In order to understand what is required by the Gateway process at the SM for building DomoBus supervision-level messages, the possible DCN Data formats are shown next as examples of relevant data and associated events. Notice that DCN Data field is a subset, with size $TLen$ - 8 bytes, of the full DCN Packet (see annexes), where $TLen$ is the packet size. [28]

- **GET** – The GET message is a request for information on the state of a device;
- **SET** – The SET message is a direct order for changing the state of a device;
- **NOTIFY** – The NOTIFY message informs the supervision application about a state modification that occurred locally. This message can only originate from the CMs;
- **EXEC** – The EXEC message allows executing a specific function on a DCN CM application.
Control-level GET/SET messages can originate from DCN CMs even though typically it is a supervision-level GET/SET message that originates from the Supervisor process at the SM, reaches the Gateway process which then generates a DCN GET/SET message that targets a DCN device that is relayed by the DCN physical gateway. In the GET or SET message:

- The $PropDescOrig$ field identifies the property that generated the packet and it is required to allow clearing the event that originated the packet;
- The $PropDescOrig$ field identifies the target property for getting or setting a value;
- The $Value$ field holds the target value on SET messages and holds 0 for GET messages but returns with the correct value.

In the DCN NOTIFY message, the $PropDesc$ field identifies the property that had a state change, and $Value$ is the new value associated to that property state change.

Applications running in DCN CMs manage their devices using variables named properties. These properties save the state of a device. Each DCN CM application potentially handles multiple properties of multiple devices. If an application needs to write or read properties of a neighbour application on the same or other CM, it needs to send DCN SET or GET messages to that application. Most of the times it is expectable that DCN CM applications do not share information (only receiving GET and SET messages originating from SMs) but there may be certain use cases where a CM implements logic between CM applications. [2]

### 3.2.1. Supervision Layer

The Supervision Layer is composed by a LAN (or WLAN) of SMs, where an SM is typically a Single Board Computer with available RAM and I/O interfaces such as USB, Ethernet, USART, SPI and I2C (e.g. Raspberry Pi, Odroid-XU4). These SMs can run several distributions of the Unix operating system which, along with their hardware resources, allow for programming complex behaviors such as performing multiple actions, system-wide, in response to a given event.

SMs are responsible for system management and maintenance of a coherent distributed house state using the DomoBus Device Model for that purpose. Every $(device, property)$ tuple instantiated on the system is saved as a state variable on one SM according to a placement metric (see section 4.1.5 for placement metric details and section 3.3 for DomoBus Device Model details). Notice that only DCN CMs also use the native DomoBus Device Model on their applications, which means in the case of DCN every $(device, property)$ tuple exists on the SM network but also locally on the CM. This allows for certain applications on DCN CMs to apply some level of control over property changes such as when to send NOTIFY messages to the SM for example. It is not expected that CMs from other vendors implement the DomoBus Device Model, since it is meant to be used by SMs. This device model is a key element for enabling the supervision layer to follow a generic approach that is independent of the different control network technologies and devices.
To be able to supervise a home with multiple control networks – that may have complex rules and a very large number of devices (i.e. super automated homes) – there can be as many SMs as needed, such as having only one for small houses or placing one per floor / per room on larger ones. They communicate with each other using UDP/IP for reduced latency and each one can connect to multiple DomoBus physical gateways (i.e. to multiple control-level networks). This approach allows for a distributed supervision, offering benefits regarding response time and reliability (i.e. no single points of failure). SMs receive information from CMs, process it according to programmed rules, modify the overall system state variables and issue the appropriate commands back to the CMs when applicable. In this way, the interaction with other systems also becomes easier allowing for interoperation and achievement of integrated solutions. [29]

![DomoBus architecture – Supervision Network emphasis](image)

One SM runs two main types of OS processes written using the C programming language:

- **Supervisor** – runs code that implements the DomoBus Supervision Model and utilizes the DomoBus Supervision Library to accomplish it. Assumes the role of Rules Engine, guaranteeing that the committed house behavior portion it received is executed correctly. It is also responsible for time management, as seen with timing sequences. It communicates with the DCOMM thread in the same process to send DomoBus Supervision Level packets to a target Gateway process (i.e. SET) or a target Supervisor process (i.e. GET, NOTIFY or EXEC).
- **Gateway** – implements the translation logic of supervision-level packets to a specific control-level format and vice versa. It is specialized in translation only, since it does not manage any packet exchanges or acknowledgments and has an inbound and outbound buffer. It communicates with the DomoBus physical gateway via USB whenever it receives supervision-level packets, translating them to a specific control-level format and proxying them downstream. It also communicates with the DCOMM thread in the same process via socket whenever it receives control-level packets, translating them to supervision-level format.

Supervision level packets are only exchanged between SM processes (notice that the PropDesc fields have the same format as shown in Figure 57):

- GET packets request the value of a device's property stored on the supervision layer, more specifically from the owner of the (device, property) tuple memory address (i.e. a Supervisor process). The GET's DevAddrW and PropDescW may be used to identify another (device, property) tuple – of the same type as the target tuple – for the value to be stored. The answer to GET is sent using an A_GET packet.
- SET packets target Gateway processes in order to set the indicated Value to DevAddr, PropDesc. The DevAddrOrig and PropDescOrig fields may be used to identify who originated the command in order to forward it an ACK or ERROR message (can be 0 if not relevant).
- NOTIFY packets target another Supervisor process in order to inform it that a (device, property) tuple of their ownership needs to be changed.
- EXEC packets target any Supervisor process in order to initiate some custom execution of a DomoBus Supervision Library function.
3.3. Device Model

DomoBus introduces a flexible and generic model for defining a device and its capabilities. This model facilitates the Supervision Layer’s ability to manage the behavior of the system, independently of the idiosyncrasies of each physical device and of the control-level network in which it operates, being fundamental to achieve a uniform abstract representation of every device.

The model assumes that each device is described by a set of properties and each property has a value. The device identifier is a 32 bits unsigned integer as seen in Figure 11. Each property represents a capability of the device, which is described by an unsigned 8 bit integer identifier. Its value can be a signed integer of 8 bits or 16 bits, an Enumerate or a DomoBus Array (i.e. 1 byte for the array length + N bytes for the data). If the property’s value size is 8 bits, it can be a normal property, or a special type of property called a “Command” (see Table 5 for more information). Every time a command property has a value attributed that is different from 0, that will trigger condition/expression checking – therefore possibly triggering actions – and afterwards the property’s value is automatically reset to 0. [30]

These are examples of instanced (device, property) tuples:

<table>
<thead>
<tr>
<th>Device Name</th>
<th>Property Name</th>
<th>Property Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TV</td>
<td>OnOff</td>
<td>Enum</td>
<td>Turns the living room TV on/off</td>
</tr>
<tr>
<td></td>
<td>Volume</td>
<td>8 bits</td>
<td>Sets living room TV volume (0 to 127)</td>
</tr>
<tr>
<td></td>
<td>Channel</td>
<td>16 bits</td>
<td>Sets living room TV channel (0 to 300)</td>
</tr>
<tr>
<td>Light</td>
<td>OnOff</td>
<td>Enum</td>
<td>Turns the main living room light on/off</td>
</tr>
<tr>
<td></td>
<td>LightIntensity</td>
<td>8 bits</td>
<td>Sets living room light intensity (0-127)</td>
</tr>
<tr>
<td>TemperatureSensor</td>
<td>Temperature</td>
<td>16 bits</td>
<td>Returns the living room’s temperature</td>
</tr>
<tr>
<td>IFTTT</td>
<td>SlackChannel1</td>
<td>Domobus Array</td>
<td>Sends msg to Slack channel over IFTTT</td>
</tr>
</tbody>
</table>

Table 2 - Devices and Properties examples

Both simple and complex devices can be represented due to the simplicity of the model, from simple temperature sensors to full-fledged media devices. These collections of properties can be read or written to, in which the latter is the way to perform an action. It is also straightforward to dynamically add new device types to the system only requiring specifying their properties and a description to aid end-user and developer interpretation. The DomoBus supervision layer interacts with physical devices through the exchange of SET messages by specifying a (device, property) tuple, the opcode of the message, and a value, and is notified of new values via NOTIFY messages originating from the control-level. The device types and device instances are specified in a generic way using XML. [30]

For further information on system specification please refer to section 3.6.
3.4. Supervision Model

The DomoBus Supervision Model implements the desired home automation mechanisms. It is executed in every Supervisor process, using the DomoBus Supervision Library to accomplish it. This chapter explains how the Supervisor process manages rules as a rules engine.

The Supervisor process is decomposed into 3 threads, one for time management, one for keyboard interaction and finally the main one that is listening to an open socket for connections from the DCOMM thread as well as managing users, devices and the ruleset in memory. This means that the Supervisor process will receive data from the DCOMM thread – after the latter has received a Supervision Level packet that targets that Supervisor process – and will process that data according to the programmed house behavior rules (which also exist on the Supervisor in memory).

At the same time a Supervisor process is actively processing activities in the house it can dynamically change its previously committed behavior. A developer can interact with a single Supervisor process via UNIX shell – through the keyboard thread – after it is launched. This allows either a developer or an automated task to program a Supervisor in real time.

The automation mechanisms implemented by the DomoBus Supervision Model are the following:

- **Simple Rule** – it is the foundation of the ruleset. It is used when only one (device, property) tuple’s value is tested. If the value of that tuple makes the condition evaluate to true, then a list of actions is executed. The possible conditions for the Simple Rule, according to mathematical notation, are =, ≠, <, ≤, >, ≥ and “DO” (where the “DO” condition implies that the list of actions is always executed every time the value of the tuple changes).

  Example:

  \[
  \text{IF } \text{light\_switch}=1 \ \text{THEN} \ (\text{left\_lamp}=1, \ \text{right\_lamp}=1) \]

  If the “light\_switch” (device, property) tuple equals 1, then the “left\_lamp” and “right\_lamp” (device, property) tuples are set to 1.

- **Expression Rule** – it is used when multiple (device, property) tuples are tested (i.e. multiple conditions). If the combined Boolean value of the conditions evaluate to true, then a list of actions is executed. The possible logic operators for the Expression Rule are EQUAL, NOT, AND, OR, XOR and XNOR.

  Example:

  \[
  \text{IF } \text{light\_switch\_1}=1 \ \text{OR} \ \text{light\_switch\_2}=1 \ \text{THEN} \ (\text{left\_lamp}=1, \ \text{right\_lamp}=1) \]

  If the “light\_switch\_1” or “light\_switch\_2” (device, property) tuples equal 1, then the “left\_lamp” and “right\_lamp” (device, property) tuples are set to 1.

- **Schedule Rule** – it is used when a single condition is used to test the time. If the target time is met, a list of actions is executed. Additionally, Schedule Rules can have different periodicities:
Single event (occurs only once), daily (hour, minute and second), weekly (day of the week, hour, minute and second), monthly (day of the month, hour, minute and second) and annual (day of the year, hour, minute and second).

Example:

\[
\text{IF } \text{time}=1566920338 \text{ (Single Event) THEN } \text{(alarm}=1, \text{ light}=1) \\
\]

If the time is equal to 1566920338 seconds, the “alarm” and “light”\( (device, property) \) tuples are set to 1.1566920338 seconds corresponds to Tuesday, August 27, 2019 3:38:58 PM.

- **Timing Sequence** – it is used for introducing a pause (in seconds) before activating a list of actions. The timing sequence is used as an action on Simple, Expression and Schedule rules.

Example:

\[
\text{sequence}_1: \text{WAIT 15s THEN door}=0 \\
\text{IF motion_sensor}=1 \text{ THEN (door}=1, \text{ START sequence}_1) \\
\]

If the “motion_sensor”\( (device, property) \) tuple value is equal to 1, then the “door”\( (device, property) \) tuple is set to 1 and then “sequence_1” starts. After 15 seconds the “door”\( (device, property) \) tuple is set back to 0.

Besides these definitions, there are two types of actions: activation actions, which are executed when a rule (i.e. it's conditions) was \textit{false} and switches to \textit{true}, and deactivation actions, which are executed when a rule was \textit{true} and switches to \textit{false}. A developer can add a mix of activation and deactivation actions to the list of actions of a rule, depending on his development strategy.

As it can be seen, the ruleset of a \textit{Supervisor} process is composed of multiple of the previous rules. These rules are themselves composed by a set of (supervision model) objects that are instantiated and linked during the programming of the rules. These objects are:

- **Device,Property Tuple** – used in all rule types, holds a value of a certain DomoBus Device Model type
- **Simple Condition** – used in simple rules, tests a single \( (device, property) \) tuple against a value
- **Expression** – expression rule, tests multiple expression conditions related by logical operators
- **Expression Condition** – used in expression rules, tests a single \( (device, property) \) tuple against a value
- **Logical Operator** – used in expression rules, relates two expression conditions’ Boolean values
- **Activation Action** – used in all rule types, assigns a new value to a \( (device, property) \) tuple
- **Inactivation Action** – used in all rule types, assigns a new value to a \( (device, property) \) tuple
- **Sequence** – timing sequence, creates a sequence of items, each one associated with an action list
- **Sequence Item** – used in timing sequences, launches an action list after \( X \) seconds have passed
- **Schedule** – schedule rule, associates a schedule period to a specific action list
- **Schedule Period** – used in schedule rules, defines a period for launching a specific action list

Linking these (supervision model) objects is achieved using commands from the DomoBus Supervision Library during the programming phase of the house behavior on a \textit{Supervisor} process (e.g. using the UNIX shell, manually or by inputting a file with the commands).
3.5. Supervision Library

These are the commands offered by the DomoBus Supervision Library in order to program the ruleset:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Command Description</th>
<th>Parameters Format</th>
<th>Action Hex ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADP</td>
<td>Add Device Property</td>
<td>(d,p,p_cfg)</td>
<td>x00</td>
</tr>
<tr>
<td>AEX</td>
<td>Add Expression</td>
<td>(e)</td>
<td>x01</td>
</tr>
<tr>
<td>ASC</td>
<td>Add Simple Condition</td>
<td>(d,p,c,op,val)</td>
<td>x05</td>
</tr>
<tr>
<td>AEC</td>
<td>Add Expression Condition</td>
<td>(d,p,c,e,op,val)</td>
<td>x09</td>
</tr>
<tr>
<td>AAC</td>
<td>Add Activation action to Condition</td>
<td>(d,p,c,a,action)</td>
<td>x0C</td>
</tr>
<tr>
<td>AIC</td>
<td>Add Inactivation action to Condition</td>
<td>(d,p,c,a,action)</td>
<td>x0C</td>
</tr>
<tr>
<td>AAE</td>
<td>Add Activation action to Expression</td>
<td>(e,a,action)</td>
<td>x0D</td>
</tr>
<tr>
<td>AIE</td>
<td>Add Inactivation action to Expression</td>
<td>(e,a,action)</td>
<td>x0D</td>
</tr>
<tr>
<td>AOP</td>
<td>Add Operator</td>
<td>(e,o,log_op,oper1,oper2)</td>
<td>x0E</td>
</tr>
<tr>
<td>SPV</td>
<td>Set Property Value</td>
<td>(d,p,val)</td>
<td>x0F</td>
</tr>
<tr>
<td>ESC</td>
<td>Enable Simple Condition</td>
<td>(d,p,c)</td>
<td>x06</td>
</tr>
<tr>
<td>DSC</td>
<td>Disable Simple Condition</td>
<td>(d,p,c)</td>
<td>x07</td>
</tr>
<tr>
<td>EEX</td>
<td>Enable Expression</td>
<td>(e)</td>
<td>x02</td>
</tr>
<tr>
<td>DEX</td>
<td>Disable Expression</td>
<td>(e)</td>
<td>x03</td>
</tr>
<tr>
<td>MC</td>
<td>Modify Condition</td>
<td>(d,p,c,op,val)</td>
<td>x0B</td>
</tr>
<tr>
<td>RSC</td>
<td>Remove Simple Condition</td>
<td>(d,p,c)</td>
<td>x08</td>
</tr>
<tr>
<td>REX</td>
<td>Remove Expression</td>
<td>(e)</td>
<td>x04</td>
</tr>
<tr>
<td>REC</td>
<td>Remove Expression Condition</td>
<td>(d,p,c)</td>
<td>x0A</td>
</tr>
<tr>
<td>RAC</td>
<td>Remove Activation action from Condition</td>
<td>(d,p,c,a)</td>
<td>x21</td>
</tr>
<tr>
<td>RIC</td>
<td>Remove Inactivation action from Condition</td>
<td>(d,p,c,a)</td>
<td>x21</td>
</tr>
<tr>
<td>RAEE</td>
<td>Remove Activation action from Expression</td>
<td>(e,a)</td>
<td>x22</td>
</tr>
<tr>
<td>RIE</td>
<td>Remove Inactivation action from Expression</td>
<td>(e,a)</td>
<td>x22</td>
</tr>
<tr>
<td>RAS</td>
<td>Remove Action from Sequence-item</td>
<td>(s,a)</td>
<td>x23</td>
</tr>
<tr>
<td>RAX</td>
<td>Remove Activation action from Schedule</td>
<td>(x,a)</td>
<td>x24</td>
</tr>
<tr>
<td>RIX</td>
<td>Remove Inactivation action from Schedule</td>
<td>(x,a)</td>
<td>x24</td>
</tr>
<tr>
<td>ASQ</td>
<td>Add Sequence</td>
<td>(s)</td>
<td>x10</td>
</tr>
<tr>
<td>ASX</td>
<td>Add Sequence Item</td>
<td>(s,item,wtime)</td>
<td>x11</td>
</tr>
<tr>
<td>MSI</td>
<td>Modify Sequence Item</td>
<td>(s,item,wtime)</td>
<td>x18</td>
</tr>
<tr>
<td>AAS</td>
<td>Add Action to Sequence item</td>
<td>(s,item,a,action)</td>
<td>x12</td>
</tr>
<tr>
<td>ESQ</td>
<td>Enable Sequence</td>
<td>(s)</td>
<td>x13</td>
</tr>
<tr>
<td>DSQ</td>
<td>Disable Sequence</td>
<td>(s)</td>
<td>x14</td>
</tr>
<tr>
<td>SSQ</td>
<td>Start Sequence</td>
<td>(s)</td>
<td>x15</td>
</tr>
<tr>
<td>CSQ</td>
<td>Cancel Sequence</td>
<td>(s)</td>
<td>x16</td>
</tr>
<tr>
<td>RSQ</td>
<td>Remove Sequence</td>
<td>(s)</td>
<td>x17</td>
</tr>
<tr>
<td>ASXP</td>
<td>Add Schedule Period</td>
<td>(x,rep_m,week_d,rep_p,begin_t,end_t)</td>
<td>x1A</td>
</tr>
<tr>
<td>AAXP</td>
<td>Add Schedule Period</td>
<td>(x,i,on_t,off_t)</td>
<td>x1B</td>
</tr>
<tr>
<td>AIXP</td>
<td>Add Inactivation action to Schedule</td>
<td>(x,i,a)</td>
<td>x1C</td>
</tr>
<tr>
<td>ESXP</td>
<td>Enable Schedule</td>
<td>(x)</td>
<td>x1D</td>
</tr>
<tr>
<td>DSXP</td>
<td>Disable Schedule</td>
<td>(x)</td>
<td>x1E</td>
</tr>
<tr>
<td>RSXP</td>
<td>Remove Schedule</td>
<td>(x)</td>
<td>x1F</td>
</tr>
<tr>
<td>MSXP</td>
<td>Modify Schedule</td>
<td>(x,b,e)</td>
<td>x20</td>
</tr>
</tbody>
</table>

Table 3 - Commands for building a Supervisor ruleset

There are a few other commands that are used for debugging, such as GPV – Get Property Value or LDP - List Devices Properties, that are not listed here for brevity. Table 4 shows the actions’ format. The “action-binary” part is any supervision library command of Table 3 written in hexadecimal format.
<table>
<thead>
<tr>
<th>Tag</th>
<th>Parameters Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC</td>
<td>app &quot;action-binary&quot;</td>
</tr>
<tr>
<td>EXEC_V</td>
<td>app &quot;action-binary&quot; index device_index,property_index</td>
</tr>
<tr>
<td>SET</td>
<td>device_origin,property_origin_device_destination,property_destination_value</td>
</tr>
<tr>
<td>SET_V</td>
<td>device_origin,property_origin_device_destination,property_destination,device_index,property_index</td>
</tr>
<tr>
<td>GET</td>
<td>device_origin,property_origin_device_destination,property_destination</td>
</tr>
<tr>
<td>NTFY</td>
<td>device_destination,property_destination_value</td>
</tr>
<tr>
<td>NTFY_V</td>
<td>device_destination,property_destination_device_index,property_index</td>
</tr>
</tbody>
</table>

Table 4 - Actions format when using Supervision Library commands

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Name</th>
<th>Size</th>
<th>Format related details</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>device address id</td>
<td>32 bits</td>
<td>0 - 4294967295, 65535 per Supervisor process</td>
</tr>
<tr>
<td>p</td>
<td>property descriptor id</td>
<td>8 bits</td>
<td>invalid: 0 ≤ val ≤ 31 is valid, else invalid: 0 ≤ val ≤ 63 → 8bit prop, 64 ≤ val ≤ 127 → 16bit prop, val ≥ 128 → domobus array prop</td>
</tr>
<tr>
<td>p_cfg</td>
<td>property type</td>
<td>8 bits</td>
<td>if prop 8bit: 0 → normal prop, else command if prop 16bit: ignored if prop domobus array: max length</td>
</tr>
<tr>
<td>c</td>
<td>condition id</td>
<td>32 bits</td>
<td>0 - 4294967295</td>
</tr>
<tr>
<td>e</td>
<td>expression id</td>
<td>32 bits</td>
<td>0 - 4294967295</td>
</tr>
<tr>
<td>a</td>
<td>action id</td>
<td>32 bits</td>
<td>0 - 4294967295</td>
</tr>
<tr>
<td>o</td>
<td>operator id</td>
<td>32 bits</td>
<td>0 - 4294967295</td>
</tr>
<tr>
<td>s</td>
<td>sequence id</td>
<td>32 bits</td>
<td>0 - 4294967295</td>
</tr>
<tr>
<td>item</td>
<td>sequence item id</td>
<td>32 bits</td>
<td>0 - 4294967295</td>
</tr>
<tr>
<td>x</td>
<td>schedule id</td>
<td>32 bits</td>
<td>0 - 4294967295</td>
</tr>
<tr>
<td>i</td>
<td>schedule period id</td>
<td>32 bits</td>
<td>0 - 4294967295</td>
</tr>
<tr>
<td>op</td>
<td>operation type</td>
<td>8 bits</td>
<td>EQ=0,NE=1,LE=2,LT=3,GE=4,GT=5,DO=6</td>
</tr>
<tr>
<td>log_op</td>
<td>logic operator type</td>
<td>8 bits</td>
<td>EQ=0,NOT=1,AND=2,OR=3,XOR=4,XNOR=5</td>
</tr>
<tr>
<td>oper1(and 2)</td>
<td>operand</td>
<td>variable</td>
<td>(1byte) operand_type(= 0), d, p, c OR (1byte) operand_type(= 1), o</td>
</tr>
<tr>
<td>wtime</td>
<td>sequence waiting time</td>
<td>32 bits</td>
<td>0 - 4294967295, in seconds</td>
</tr>
<tr>
<td>action</td>
<td>action</td>
<td>variable</td>
<td>action as specified in Table 5</td>
</tr>
<tr>
<td>action-binary</td>
<td>(any supervision library command in hexadecimal format)</td>
<td>variable</td>
<td>1st hexadecimal number (action hex id) has the code of the command to execute, remaining numbers are the command's parameters</td>
</tr>
<tr>
<td>val</td>
<td>value</td>
<td>variable</td>
<td>size depends on property/condition type</td>
</tr>
<tr>
<td>rep_m</td>
<td>repetition mode</td>
<td>8 bits</td>
<td>0=once,1=daily, 2=weekly, 3=monthly, 4=yearly</td>
</tr>
<tr>
<td>rep_p</td>
<td>repetition period</td>
<td>8 bits</td>
<td>every N days/weeks/months/years</td>
</tr>
<tr>
<td>week_d</td>
<td>weekday</td>
<td>8 bits</td>
<td>0 - 6 (0 is Sunday, 6 is Saturday)</td>
</tr>
<tr>
<td>begin_t</td>
<td>begin time</td>
<td>32 bits</td>
<td>e.g: 30-10-2019 10:30:15 (0 = now)</td>
</tr>
<tr>
<td>end_t</td>
<td>end time</td>
<td>32 bits</td>
<td>e.g: 31-10-2019 00:00:00 (0 = never ends)</td>
</tr>
</tbody>
</table>

Table 5 – Detailed format of Supervision Library commands’ parameters

The last table gives information on how the Supervisor parser will read the commands’ parameters. They will always be translated to binary as the final format. Here is an example of usage of Supervision Library commands as manual/file input on the Supervisor shell, in order to add actions to an existing expression. These execute an SPV command on (dev, prop) (22,22) setting it to 1 and 0, respectively:

AAE 6 1 EXEC 1 "x0F\x16\x00\x00\x00\x16\x01"
AIE 6 2 EXEC 1 "x0F\x16\x00\x00\x00\x16\x00"
3.6. System Specification Language

A DomoBus system is entirely described using an XML-based specification language. The full system specification resides in a single file to simplify access and management of the information and is capable of being efficiently extended due to the nature of the utilized language. The file can be logically divided into two parts: [30]

- Definition of generic device and property types
- Definition of RBAC, system preferences and of the physical structure of the home or building, including instanced devices

Firstly, there are a few XML conventions that need to be considered when writing the system specification file: [30]

- Preferred use of attributes over elements due to achieving a more compact format
- A dedicated element will always be used to indicate the beginning and ending of list
- Every element will require an unsigned integer attribute as an ID
- An element can reference another element by using an attribute that starts with “Ref” such as RefElement1 = “123”

Below are the various elements that can be used inside a DomoBus System Specification XML file: [30]

- List of Device Classes – list of device classes for referencing when declaring device types

```
<DeviceclassList>
  <DeviceClass ID="1" Name="Lighting"/>
  <DeviceClass ID="2" Name="Motion"/>
  <DeviceClass ID="3" Name="Heating"/>
</DeviceclassList>
```

Figure 13 - List of Device Classes on the DomoBus System Specification XML file

- List of Conversion Formulas & Objects – list of conversion formulas and objects to allow translation of integers and byte arrays to other types of values such as floating points with double precision, signed fixed point numbers or any other type of data

```
<ConversionFormulaList>
  <ConversionFormula ID="1" Name="Triples the value" UserToSystem="3\*x" SystemToUser="x/3" DecimalPlaces="0"/>
</ConversionFormulaList>

<ConversionObjectList>
  <ConversionObject ID="1" Name="Intensity - lux" UserToSystemObj="1" SystemToUserObj="2" DecimalPlaces="2"/>
</ConversionObjectList>
```

Figure 14 - List of Conversion Formulas & Objects on the DomoBus System Specification XML file

Notice that conversion objects require that a specialized application is present on the SM in order to declare the conversion objects, in this case, with id “1” and “2” – that have complex conversion algorithms – and run them. This feature was not yet implemented by the time of this writing, but the XML specification predicts its usage.
• Lists of Value Types – the value of a property can be of three different types:
  o ScalarValueType – integer value that can be represented by 8, 16 or 32 bits.
  o EnumValueType – enumerated pair \((name, value)\), where \(value\) is 8 bit
  o ArrayValueType – array of bytes

```xml
< ScalarValueTypeList>
  < ScalarValueType ID="1" Name="Power" NumBits="16" Units="Watt" MinValue="0" MaxValue="10000" Step="10"/>
  </ ScalarValueType>
</ ScalarValueTypeList>

< EnumValueTypeList>
  < EnumValueType ID="1" Name="Light" NumBits="16" Units="Lux" MinValue="0" MaxValue="32767" Step="1"/>
  </ EnumValueType>
</ EnumValueTypeList>

< ArrayValueTypeList>
  < ArrayValueType ID="1" Name="Float IEEE" MaxLen="8">
    <ValueConversion Type="OBJECT" Ref="5"/>
  </ ArrayValueType>
</ ArrayValueTypeList>
```

*Figure 15 - List of Value Types on the DomoBus System Specification XML file*

• List of Device Types – list of device types characterized by a collection of properties

```xml
< DeviceTypeList>
  < DeviceType ID="1" Name="Adjustable Light" RefDeviceClass="1" Description="-">
    < PropertyList>
      < Property ID="1" Name="On-Off" AccessMode="Rh" ValueType="ENUM" RefValueType="1"/>
      < Property ID="2" Name="Intensity" AccessMode="Rh" ValueType="SCALAR" RefValueType="1"/>
    </ PropertyList>
  </ DeviceType>
</ DeviceTypeList>
```

*Figure 16 - List of Device Types on the DomoBus System Specification XML file*

• List of Access Levels – list of access levels that are then attributed to users in order to enable them to execute actions over allowed devices

```xml
< AccessLevelList>
  < AccessLevel level="1" Name="Guest"/>
  < AccessLevel level="2" Name="Common User - Child"/>
  < AccessLevel level="3" Name="Common User - Parent"/>
  < AccessLevel level="4" Name="Administrator"/>
</ AccessLevelList>
```

*Figure 17 - List of Access Levels on the DomoBus System Specification XML file*

• List of Users – list of users with an associated access level

```xml
< UserList>
  < User ID="1" Name="John" Password="[hash]" AccessLevel="3"/>
</ UserList>
```

*Figure 18 - List of Users on the DomoBus System Specification XML file*
• House Structure – specifies floors and divisions

```xml
<House ID="1" Name="House1" Address="abed" Phone="12345678">
    <FloorList>
        <Floor ID="1" Name="Ground Floor" HeightOrder="0"/>
        <Floor ID="2" Name="First Floor" HeightOrder="1"/>
    </FloorList>
    <DivisionList>
        <Division ID="1" Name="Hall" RefFloor="1" AccessLevel="3"/>
        <Division ID="2" Name="Kitchen" RefFloor="1" AccessLevel="3"/>
        <Division ID="3" Name="Bedroom" RefFloor="2" AccessLevel="3"/>
    </DivisionList>
</House>
```

*Figure 19 - House Structure on the DomoBus System Specification XML file*

• List of Services – a service groups devices that are related functionally. A device can join multiple services

```xml
<ServiceList>
    <Service ID="1" Name="Heating"/>
    <Service ID="2" Name="Lighting"/>
    <Service ID="3" Name="Security"/>
</ServiceList>
```

*Figure 20 - List of Services on the DomoBus System Specification XML file*

• List of Instanced Devices – list with all the instanced devices on the system

```xml
<DeviceList>
    <Device ID="1" RefDeviceType="1" Name="Kitchen Lamp" Address="#0100" RefDivision="1" AccessLevel="3,3" UserBlocked="-2">
        <DeviceService RefService="1"/>
        <DeviceService RefService="2"/>
    </Device>
</DeviceList>
```

*Figure 21 - List of Instanced Devices on the DomoBus System Specification XML file*

• List of Device States – list that has the initial state of any \((\text{device, property})\) tuple. The `InvalidValue` field states that the current property’s value is unknown or there is some error.

```xml
<DeviceStateList>
    <DeviceState RefDevice="1" RefProperty="1" Value="0" InvalidValue="FALSE"/>
    <DeviceState RefDevice="2" RefProperty="1" Value="0" InvalidValue="FALSE"/>
</DeviceStateList>
```

*Figure 22 - List of Devices States on the DomoBus System Specification XML file*

• List of Scenarios – list of action aggregates for triggering upon a certain conditional expression

```xml
<ScenarioList>
    <Scenario ID="1" Name="Shut Living Room Lights Off">
        <ActionList>
            <Action ID="1" RefDevice="1" RefProperty="1" Value="0"/>
            <Action ID="2" RefDevice="2" RefProperty="1" Value="0"/>
        </ActionList>
    </Scenario>
</ScenarioList>
```

*Figure 23 - List of Scenarios on the DomoBus System Specification XML file*
4. Developed Work

4.1. DHM Concepts

4.1.1. Domobus Automation Blocks

The Domobus Automation Block (DAB) is a mechanism defined for Domobus House Manager (DHM) usage that allows the specification of complex Domobus automation rules. DABs are defined in a generic way, acting like automation templates, which can be easily re-used and instantiated in real systems using the DHM. At the moment, only the DHM is able to process DABs. As part of the workflow detailed in 4.2.1, the DHM will aggregate all rooms’ DABs and convert the final aggregated DAB – called the final house behavior – to a Domobus Assembly file composed of Supervision Library commands. A Supervisor process receives the assembly file, interprets the Supervision Library commands and creates Supervision Model objects that constitute the ruleset (i.e. simple rules, expression rules and timing sequences). This section describes the concepts associated with Domobus Automation Blocks and introduces the language used in their definition, which allows the creation of a database of block templates.

4.1.1.1. Automation Blocks Definition

An automation block is a functional block that may have multiple inputs, outputs and virtual properties. Each one of these is identified by an alphanumeric string ID, which represents a (dev,prop) tuple and therefore inherits the tuple’s property type. One of the main characteristics of automation blocks is that they can interconnect to form more complex behavior. One output can connect to multiple inputs of the same type. After connecting multiple blocks, a single block is created to reflect the aggregations. In that resulting block, inputs and outputs may be left unconnected and will remain in it so, ultimately, the user may (or not) connect hardware devices to them in the instantiation phase. This is useful as automation blocks should be designed as generic and flexible as possible and, in some situations, sensors may not be available to connect to some inputs. To support inputs that have no properties connected to it, inputs can have a static value assigned, which will be considered as the input’s result for evaluating logical expressions. Conversely, outputs that were connected to inputs always get converted to virtual properties. So, at the user level, it can be said that e.g. a temperature sensor connects to input 1, a light intensity sensor connects to input 2 and a light connects to output 1. However, at the Supervisor process level, where the automation block gets translated into rules and executed, the meaning of the property is not relevant; only its type is, which may be 0 (8 bits), 64 (16 bits), 128 (32 bits, not implemented) and 192 (Domobus Array). As mentioned in 3.3, besides the property type, any input, output or virtual property can be a command. Properties have an initial value (and an array size in case it is a Domobus array) and the possibility to mark an output/virtual as a state variable (along with their cardinality).
4.1.1.2. Expansive Inputs

An important concept is the Expansive Input, which accepts multiple connections (from properties of the same type). It is defined in basis DABs and requires the definition of an expression composed by a condition, operator and wildcard (%X), e.g.:

\[ I_1 \geq 5 \text{ AND } %X \]

Assuming the ID of the declared (generic) expansive input is \( I_1 \), and that 3 hardware devices are connected to \( I_1 \), this expression expands to the following \textit{global scope} expression after instantiation:

\[ I_1 \geq 5 \text{ AND } I_2 \geq 5 \text{ AND } I_3 \geq 5 \]

\( I_1, I_2 \) and \( I_3 \) now correspond to each of the 3 introduced hardware devices. In case outputs (instead of hardware devices) connect to inputs, an expansion to \textit{virtual} properties would appear instead of to new inputs, while the old outputs and the (generic) expansive input would be deleted. Normally, all the 3 inputs would have to be processed individually and correlated manually. This would make the solution more complex and less general. Using the expansion feature of a block’s input, one just has to specify which properties connects to it and they will be processed automatically, making it very easy to add or remove properties from an input. For example, if we are connecting multiple pushbutton switches to an input, it is enough that one is active to make the input active. So, the operator OR will be used. If we want to detect that there is low light in a room and we are using various light intensity sensors, one can test if each of them is less than 40 (for example), and we will only consider it is dark if all conditions are true at the same time. So, in this case, the correct aggregation operator to use is AND. Regarding outputs, these can have multiple hardware devices connected to them and in the \textit{final} DAB that output is expanded to that nº of connected devices.

\[
\text{Figure 24 - Example of Expandable Input}
\]
4.1.1.3. Finite State Machines

To offer the required flexibility, a block can internally contain multiple deterministic finite automata, a type of finite state machine where the transition from a state is to a single particular next state, for each input symbol. A state is defined by a subset of the block’s outputs and virtuals marked as state variables and has a unique ID (i.e. Sx, where x is an incremental number, starting at S1). This means it is also possible for a block to have outputs (and virtuals) that are not state variables (i.e. are not part of any state machine). A state machine is defined by their own set of state variables and have only one active state at a time. A block without states corresponds to a functional block – with no state machines – that implements pure combinatorial logic, i.e. where outputs depend exclusively on the current values of the inputs. A block with a set of states that have transitions between them – implementing state machines – corresponds to a functional block that implements sequential logic, i.e. where the outputs may reflect past occurrences of input values. There can be a mix of both logics in a block. The internal structure of a block allows the implementation of state machines of type Mealy. In these machines, the outputs depend on the current state and also on the inputs, i.e. one may have different outputs for the same state depending on what transition was fired. The implementation of Moore state machines is also supported, where the outputs are directly associated with the states, i.e. each state has a fixed output. During the first experiments, thinking in terms of Moore machines seemed to simplify the design process, although this is not, at all, mandatory.

Each block’s state may have multiple transitions to and from it. A transition corresponds to one DomoBus expression rule, which is an association of one logical expression – that tests the block’s (dev,prop) tuples – with multiple activation actions (transitions don’t use inactivation actions). Due to a currently existing limitation in the supervisor, which is that a schedule condition cannot be mixed with other conditions, we opted that the transition expression cannot be a schedule. If the transition expression is composed by a single condition, then the transition will be parsed as a DomoBus simple rule, otherwise it will be parsed as a DomoBus expression rule, as seen in section 3.4. An action can be one of the following types: assignment, modify rule, start timing sequence, cancel timing sequence, enable transition, disable transition, action reference. Actions will be sent as EXEC packets in the DomoBus system, except for “modify rule” actions which are sent as EXEC_V packets. The action binary part of the EXEC packet will differ depending on the type of action: “Assignment” actions are translated to SPV commands; “Enable/Disable Transition” actions are translated to EEX/DEX commands; “Start Sequence” and “Cancel Sequence” actions are translated to SSQ and CSQ commands, respectively; “Action” references will ultimately represent an assignment or enable/disable transition action. The action binary part of the EXEC_V packet for “Modify Rule” actions are MC or MSI commands, where a (dev, prop) value is passed to the modify command. “Modify Rule” actions do not appear on the DHM GUI because they are only automatically used on parameters. All the (dev,prop) tuples used as inputs, outputs and virtuals inside a block can be tested directly by the expressions present in its transitions, while only outputs and virtuals can have values assigned by actions. Notice that, in a DAB, a transition may or may not cause a block’s state to change. It depends if the transition’s actions change the value of any outputs or virtuals marked as state variables. Those are state transitions, which are used only in the context of state machines. Combinatorial transitions, on the other hand, use any inputs and global scope items but can only reference/act upon any non-state variable outputs & virtual properties.
Two very important aspects of state transitions on DABs are that, within a state machine, 1) no two transitions can fire at the same time and 2) only a subset of its transitions is able to fire depending on the current state. At the generation of any basis DAB, there is an automatic step executed by the DHM which is the addition of dedicated “enable/disable transition” actions on all state transitions. A state transition X, of state machine M1, has a source state SS and target state TG. X will automatically receive:

- one “Enable Transition Y” action for each transition Y whose source state is X’s target state TG;
- one “Disable Transition Z” action for each transition Z whose source state is not X’s target state TG (i.e. all the other transitions of that state machine).

Given the state machine of Figure 25, depending on the current state, only a subset of the transitions can fire. Assuming state 1 is the current state, then T1 and T4 are enabled and all the other transitions are disabled. T1 has its user defined actions but also has the automatically assigned “enable:T2”, “enable:T6”, “disable:T1”, “disable:T3”, “disable:T4” and “disable:T5” actions. Similarly, T4 has the automatically assigned “enable:T3”, “enable:T5”, “disable:T1”, “disable:T2” and “disable:T4” actions.

As previously mentioned, no two state transitions of the same state machine can fire at the same time. This is because, as explained in section 4.1.5, a state machine will exist in a single supervisor process, and each supervisor queues (dev, prop) tuple value changes. This means that when a (dev, prop) tuple changes value, it triggers a verification on all expressions that reference that tuple (on that supervisor only). For each one of those expressions that evaluate to true, the supervisor will sequentially execute all associated actions before moving on to the next expression. When it finishes all of those expressions, it goes back to the queue to get the next (dev, prop) tuple whose value changed. This guarantees no interleaving issues (i.e. that no two transitions fire at the same time or out of order) since the order of (dev, prop) actuation is maintained and no two expressions are evaluated at the same time. When a (dev,prop) tuple value changes which would make expression E evaluate to true, but E is disabled (i.e. its transition is disabled), the verification of E is discarded and not put into any queue.

As a final remark regarding the type of action that handles state variables, taking Figure 25’s example again, if T1’s action on the state variable is of type “Assignment”, then the automatic “disable/enable transition” feature is the same as mentioned above. If it is of type “Start Timing Sequence” then T1 receives “disable:T1”, “disable:T2”, “disable:T3”, “disable:T4”, “disable:T5” and “disable:T6” actions but the sequence item automatically receives “enable:T2” and “enable:T6” actions. This means that, when T1 fires it will execute all user defined actions but the actuation over the state variable is delayed by the timing sequence. Therefore, all transitions must be disabled until the sequence delay expires and the sequence item actions are fired. When the sequence item actions fire, they re-enable T2 and T6.
4.1.1.4. Global Scope

Additionally, a block may have global scope expressions, actions and timing sequences declared. These are not associated to any transitions; global scope expressions are referenced by other global scope expressions or by transitions’ expressions and cannot reference state variables; global scope actions are referenced by other global scope actions or by transitions’ actions and cannot change state variables’ values. Timing sequences are always declared as global scope and then used within global scope actions or state/combinatorial transition actions. Notice that only transitions originate the Supervision Library commands that constitute an Expression Rule, such as AEX, AEC, AAC, AOP. In the DomoBus assembly file, a Global scope expression will originate a new (dev,prop) tuple to represent it. Then an activation action will be added to that global scope expression in order to SPV the previous tuple to value “1” and an inactivation action will also be added to that global scope expression in order to SPV the previous tuple to value “0”. Finally, the previous tuple will be added in an expression condition, related by an AND operand, to the transition’s expression. Global scope actions will be ultimately expanded into transition’s actions when referenced in them. If they are not referenced, in the assembly file they will not be translated to anything.

4.1.1.5. Basis DAB

There are multiple types of DABs, namely basis, aggregated, and final. A basis DAB is, as the name suggests, used as the basis for more complex behaviors which can be achieved through aggregation of multiple basis DABs. This type of block usually contains a solution for a specific problem, such as logic for stairs light control or logic for motion sensor activated doors with timers, for example. The reference architecture for a basis DAB includes a set of good practices that a developer should aim for, when using the DHM, to create a standardized automation block for efficient posterior aggregation.

Figure 26 - Reference architecture for the basis DAB
A basis block built according to the reference architecture shown in Figure 26 follows these guidelines:

1) Decide if the block will have only a state machine, only combinatorial logic, or both. Notice there is no restriction for creating blocks with more than one state machine or even with just purely combinatorial logic (i.e. no state machines), in case the design approach involves simpler and lower level logic such as multiple AND gates, XOR gates, etc.

2) To implement any sequential logic via a single state machine, follow the Moore approach. Declare one of the following:
   A. Output State Variable if it is intended to expose the state machine’s state as an output of the block for interconnecting it with inputs from other blocks;
   B. Virtual State Variable if that exposure is not intended;

3) Create the states of the state machine and set state variable values that represent each state;

4) Create the (normal) outputs of the block that are uniquely related with the state machine. These outputs will be actuated only by state transitions of a single state machine and should represent important values related to the purpose of the state machine. For example:

   (Normal) output 1 (O1) is LightIntensity (0-255), input 1 (I1) is RotatingButton (0-127) and there are 3 states (SVar=1, SVar=2, SVar=3). Depending on the current state, if I1 ≤ 42 then a transition may fire which sets O1 = 80 and SVar = 1; if I1 ≥ 43 AND I1 ≤ 85 then a transition may fire which sets O1 = 160 and SVar = 2; if I1 ≥ 86 then a transition may fire which sets O1 = 255 and SVar = 3. These outputs will be considered protected and can be used for direct connection with hardware devices, including protected ones (see section 4.1.2).

5) Create replicas of the outputs of step 4 if it is intended to expose these outputs on the block for both directly connecting hardware devices to them and interconnecting them with inputs from other blocks. Replication can be efficiently achieved by adding a specific combinatorial transition per target output, as shown in section 4.3.2.

6) Create the inputs related with the state machine. These should be expansive inputs whenever possible, to simplify the definition of the DAB, and normal inputs for single instanced data.

7) Create a specific input for enabling or disabling the complete block, which should be included in all transitions’ expressions, using the AND operator with the remaining operands.

8) Create as much global scope items as possible to avoid declaration of complex expressions and actions, including sequences that will be used in state transitions’ actions.

9) Create state transitions, state transitions’ expressions & state transitions’ actions, to implement the state machine’s logic, using previously generated properties and global scope items.

10) Implement any purely combinatorial logic (i.e. that doesn’t involve changing the value of state variables) using combinatorial transitions, global scope items and new properties. Notice that combinatorial transitions’ actions may alter a (normal) virtual property that is used in a state transition’s expression, making that combinatorial transition a trigger of the state transition.

11) Create any messages on transitions that constitute important notification points.

12) Create specific inputs for parameters and then create the corresponding parameters in order to modify transitions’ behavior in real time.
4.1.1.6. Aggregated DAB

An aggregated DAB is the result of two mechanisms applied to multiple basis or aggregated DABs placed on the aggregation canvas of the DHM. The mechanism that is first applied is fusion of inputs & outputs, followed by the second mechanism, which is union of all DAB elements. Note that two separate DABs may have elements with equal IDs, since each had their own scope at creation/aggregation time. The purpose of aggregating blocks is to obtain complex behavior, which will ultimately culminate in multiple related and unrelated state machines and combinatorial logic, for usage in a specific room.

Fusion of inputs and outputs

Given a block A with input I1 and output O1, and a block B with input X1 and output Y1, if the two blocks are interconnected by an (O1, X1) connection then the resulting aggregated block is composed by input I1, output Y1 and virtual property sys_1 (i.e. the fusion of O1 and X1). The fused virtual property sys_1:

- inherits the command and info attributes of the input property;
- inherits the nstates attribute of the output property (if it exists);
- is attributed dev and prop integers for (device, property) tuple identification;

References to O1 and X1 are replaced by sys_1 in all expressions, actions and sequences. Relation of state machines between two DABs occurs when a block B, whose output K is a state variable, connects to input W of block C, and W triggers state transitions in C. Block B's output K may connect to N inputs (e.g. on different blocks), creating N fused virtual properties and, potentially, N state machine couplings.

State machine relationship is the synchronous firing of transitions of different state machines, initiated by a trigger state transition. The trigger and target state transitions are, respectively, identified by:

- the presence of a fused virtual state variable (F) in an action of the triggering state transition;
- the presence of a fused virtual state variable (F) in the target state transition’s expression;

This relation is unidirectional, meaning that the target state transition ST2 of state machine SM2, which is triggered by the trigger state transition ST1 of state machine SM1, is not an implicit trigger of ST1. Only if an action of ST2 acts on the fused virtual state variable of SM2 which in turn is referenced on ST1's expression (due to cyclic connection between two blocks). Notice that a trigger transition may be a combinatorial transition, which will not constitute a state machine relation in that case.

![Figure 27 - Example of DAB Fusion](image-url)
Union of all DAB elements

Fusion of inputs and outputs creates a path of relations between blocks. All paths (and blocks with no connections) that are placed in the aggregation canvas will be parallelly included in the resulting aggregated DAB, expanding its purpose. As such, we can look at an aggregated DAB as set of unrelated paths. This is accomplished by the union of all DAB elements, which addresses the following problems:

- The problem of conflicting IDs is resolved when a new block is placed in the aggregation canvas. All elements’ IDs of the new block will be compared with IDs already existing on the canvas and any collisions are substituted by a unique ID as well as all references to that collision.
- In the aggregator tab, selecting a DAB in the “Available Behaviors” list and clicking the “Pull Origins” button, will place the blocks that originated that aggregated DAB in the canvas. This is possible because all DAB XML files are hashed when inserted in the canvas, and each element gets a “hash” attribute that will correspond to the origin DAB. “hash” and “hashconn” elements are introduced in aggregated DABs to identify the DAB from where each element originated.

![Figure 28 - Example of DAB Union](image)

### 4.1.1.7. Final DAB

A final DAB is the result of an instantiation process which depends on the following:

- The description of the user’s system on the loaded DomoBus System Specification XML file;
- The connections between room DABs, done in the Connections tab;
- Association of (dev,prop) tuple ids of hardware devices to inputs and outputs of room DABs.

In addition to the inherited XML elements, the final DAB introduces two new elements, the “hardwareconnection” and “roombehavior”. The first one makes the association of a hardware device name to an input ID, while the second one makes the association of a room name to a DAB hash. This is used to set the DHM status when loading the final behavior. When the “Generate House Behavior” button on the top navbar is pressed, an aggregation is executed between all rooms’ DABs and all properties receive ids so they can be instantiated. This originates two files associated with each other: a DAB of type final and its corresponding assembly file. These two files are used when the “Load House Behavior” button on the top navbar is pressed. The assembly file is pushed to Supervisors and the final DAB is loaded into the DHM, putting the DHM with the status it had when the final DAB was generated.
4.1.2. Protected Outputs and Critical Services

The purpose of inputs and outputs is to connect DABs together and, ultimately, to connect hardware devices to them in the final DAB. It is easy to conceive a situation where, for example, a (normal) output actuated by one state machine is supposed to be connected to an LED for well-defined lighting scenarios. Nevertheless, while modifying the DAB that contains that state machine, a few combinatorial transitions were set to also actuate on that output, potentially disrupting the unambiguous and well-marked values for that output, that should be uniquely managed by the state machine. Notice that, since inputs can never be actionable by transitions’ actions – can only be used in expressions or as the right-side operand of assignments – they are always protected from unpredictable actuations. Also, outputs that are state variables are always protected due to state variables being fully managed by the DHM, i.e. they are only actuated by actions automatically inserted in state transitions of that state machine. A protected output is a property uniquely related with one state machine. In other words, in order to be classified as a protected output, it must only be actuated by state transitions of a single state machine. By setting the “Protection” attribute with the value "critical" on a “Service” element of the system specification XML, the user is declaring a DomoBus Service that, when associated to a Device also in that XML (see section 3.6), tags that device as protected, showing up on the Device tab of the DHM as protected as well. This classification of an output as protected will allow hardware devices, as declared in the DomoBus System Specification XML, to require being connected to predictable and controlled protected outputs. For instance, users may want to reduce risks with security related devices such as alarms and door locks or important actuators such as gas valves and water supply valves; or they may simply want to make sure high power consumption appliances such as refrigerators and air conditioners follow well defined states for guaranteed power savings. This verification is done in the Devices tab while associating devices to outputs. If a hardware device is tagged as critical and is connected to an unprotected output, a popup will be shown to decline that association.

4.1.3. Messages to External Applications

In the future, DomoBus may introduce other applications other than e.g. supervisor and control network gateway processes. In order to have a way to interact with external applications that are not necessarily related with automation, the concept of Messages was introduced. Typically, these external applications are DomoBus Gateway-like processes running on an SM, that handle the DomoBus Supervision Level protocol and also all the communication with the external entity. The approached use case for messaging with external applications was user notification, e.g. the user receives a notification by SMS on its mobile phone, by email or by any other common channel. We associate the Supervision Level process ID of the external application – which has an associated IP address, made available to the DCOMM threads – to a channel name. Each message is associated with a transition (tagged as available for notifications) and a channel (i.e. the external application). When the transition is triggered, the message is sent to the respective channel. Transitions that are be tagged with the string "[M]" on its Info attribute (i.e. their description) will be marked as available and will be placed on the list of “available transitions” of the Messages tab. A message is saved, as written by the user in the Messages tab, in a (dev, prop) tuple of
type DomoBus array and is identified as a virtual property in the final DAB. A channel is also represented by a \((dev, prop)\) tuple, uses the reserved device numbers 32765 and 32766, is also identified as a virtual property in the final DAB but it is a command. Assuming a message property ID “sys_msg_1” and a channel property ID “sys_channel_1”, the transition associated with the message automatically receives the assignment action “sys_channel_1=sys_msg_1” which is a form of value passing from one \((dev, prop)\) to the other. An assignment action with value passing is achieved using an EXEC_V (described in section 3.5) on the AAE command. Notice that sys_channel_1 is a command, therefore its value returns to 0 after it is delivered to the external application. The message is sent to the external application using the DCOMMM thread (which knows the external application IP address). DomoBus arrays are also used for the exchange of packets directly from/to physical devices (e.g. when reading a multi-vector sensor such as an accelerometer or actuating an LED array), not only messages. Messages can only be defined when creating the final DAB on the DHM Messages tab, because information about the existing channels is bound to the DomoBus System Specification XML file, since its dependent with the number of integrations the system of a specific user has.

4.1.4. DHM Behavior Parameterization

Parameterization in the DHM context is the act of modifying a rule with a \((dev, prop)\) tuple’s value, in real time, by the user. We refer to that \((dev, prop)\) tuple as the Parameter and to the modifiable objects in the rule as Parameterizable Elements. Currently, the parameterizable elements are:

- The test value of an expression condition;
- The delay value of a timing sequence item.

The parameter tuple can be any existent DAB input, output or virtual property, and it is chosen on the Parameters tab for editing the final DAB or on the Behavior Builder tab → Parameters sub-tab for editing a basis DAB. Still, as mentioned in Figure 26, the reference architecture for a basis DAB includes inputs that are used as parameters, so typically parameters will be available as DAB inputs that the user may connect to e.g. multi-level switches, keypad or any other device that allows manually changing values.

Each parameterization entry originates a combinatorial transition whose expression is a single DO condition, “do:P1”, where P1 is the alphanumeric ID of the property, and a “Modify Rule” action that uses P1’s value to change that rule. Therefore, when the property classified as parameter, P1, changes value, the action will modify a certain expression condition or sequence item delay using P1’s value. The “Modify Rule” action is an EXEC_V packet whose action binary is an MC or MSI command that targets an expression condition or sequence item delay, respectively, and where P1’s value is passed as the new value of that modification. This allows for real time modification of the committed house behavior. For example, assuming an expression E has two conditions, Lux_Value>300 AND Lights_On==1, a parameter P1 modifies E’s condition “Lux_Value>300”. Therefore, 300 is used an initial value and when P1 changes, it is substituted by P1’s value, “Lux_Value>P1”.

48
4.1.5. Behavior Distribution

A DomoBus distributed system is composed by multiple SMs, and therefore, by multiple Supervisor processes, which own certain (dev, prop) tuples, expressions, actions, sequences and others (e.g. sequence items, simple conditions, operators) in local memory. The IDs of all these objects are global and unique in a DomoBus distributed system. The final house behavior to be committed to a DomoBus system with a single Supervisor process takes the form of a single assembly file with Supervision Library commands. In case of a distributed DomoBus system with N supervisor processes, that single assembly file must be divided into N assembly files with specific programming for each supervisor process.

The problem that arises with a distributed system is finding the best placement algorithm for all of these objects, given that (dev, prop) tuples should not be copied between supervisor processes due to their unique representation of hardware device capabilities or unique virtual properties. Still, in practice, all other objects may be copied between assembly files and still be assigned unique, global IDs.

It is very important to understand some current restrictions, related with the supervisor implementation, that influence the distribution strategy. Firstly, a supervisor process A can reference all objects in its memory space but can only reference one type of object that is not in it, which is the (dev, prop) tuple (as explained in section 3.2.1). It cannot reference, e.g. expressions or actions that are not owned by it. An “EEX 5” command executed by supervisor process A would throw an error if the expression with ID “5” was on another supervisor process. Secondly, a publish-subscribe mechanism is still being implemented in order for supervisor B to be alerted that some (dev, prop) tuple’s value on supervisor A changed. With this mechanism, a GET packet would be sent from B to A in order for A to verify if any of its expressions evaluate to true with the new value.

Notice there was no functional DCOMM library at the time of this writing, so it was not possible to test the interaction between supervisor processes, especially due to the publish-subscribe mechanism not being implemented. Therefore, this section presents a first theoretical approach to the algorithm for object placement in supervisor processes.

In this context, the definition of “best placement algorithm" is the one that minimizes the total number of communications between SMs and maximizes how spread these objects are through all SMs. The objective is to have minimum latency and load balancing, while still retaining a certain degree of spatial locality for efficient fault tolerance. To simplify this analysis, the following are considered as a single interaction between SMs:

- GET followed by an A_GET packet;
- Publish-subscribe notification followed by a GET and A_GET packet;
- SET packet;
- NOTIFY packet;
- EXEC packet;

We should first look at a centralized approach, which is ineffective in each of the main optimization vectors. Given N supervision modules and X unique DAB objects (i.e. the exact number of objects as declared in the DAB XML file, without copies) then:
- **Place all X objects in SM 1.**
  SM 1 has all objects while all other SMs act as a proxy between SM 1 and each control network gateway they host. Still, this approach clearly does not minimize the number of exchanged packets since most NOTIFY packets will need to hop to SM 1 from other SMs and most SET packets will need to hop from SM1 to other SMs. Also, it is not fault tolerant or load balanced because all objects are centralized in one SM.

Copying all objects, except tuples, to all SMs is also not an option because a change in a \((dev, prop)\) that is referenced by multiple copies of the same expression can trigger multiple copies of the same action, which will flood the supervision layer with replicated SET packets. We start by identifying dependencies between objects, which aids in maintaining a good degree of spatial locality as well:

- A transition will be translated to a DomoBus expression rule. Each expression rule exists entirely in only one supervisor (i.e. it is non-distributable) and is composed by a single declaration of an expression, multiple expression conditions, multiple operators and multiple declarations of activation actions. This implies that **all objects that compose a transition must be placed in the same supervisor process.** Those are the transition’s expression and transition’s actions.
- A state machine dictates that a certain group of transitions are related by a group of states (or conversely, a group of states are related by a group of transitions). Since there are no objects in the DomoBus supervisor process to represent a state, this relationship is achieved by enabling and disabling transitions. “Enable/Disable Transition” actions reference an expression ID, therefore, according to the limitation identified regarding referencing unowned expressions, a state machine must be entirely placed in the same supervisor process.
- A global scope expression will be decomposed into one activation and one deactivation action that executes an SPV on a \((dev, prop)\) tuple that represents that global scope expression. That \((dev, prop)\) tuple will then be referenced on transitions’ expressions. This means that a **Global Scope** expression and its related \((dev, prop)\) tuple will need to be placed together in the same supervisor process.

At this point we have the following “items” to decide their placement:

- State machines (i.e. groups of state transitions);
- (Individual) combinatorial transitions;
- \((dev, prop)\) tuples related with inputs/outputs/virtuals;
- **Global Scope** Expressions and their related \((dev, prop)\) tuples;

Please recall that **global scope** actions exist only on the context of the DAB to be used inside transitions’ actions, so they don’t need to be taken into consideration for placement in assembly. **Global scope** sequences are replicated on all assembly files, so they do not need to be accounted for placement. For example, in a DAB, if two transition actions reference the same **global scope** sequence and they are placed in different supervision processes, then that sequence will be instantiated twice, one time in each assembly file. The referenced IDs of each sequence will be replaced in each corresponding file.
In order to achieve the best placement defined above, we must understand how the “items” enlisted for placement are related, when they are placed in different SMs. A transition is related with a \((\text{dev}, \text{prop})\) tuple placed in another supervisor process if it references that tuple in either one of the following:

- Its expression. It will have a cost of one interaction: a GET packet is sent to the owning supervisor process and the corresponding \(A_{\text{GET}}\) packet is received;
- Its actions. It also has a cost of one: a NOTIFY packet is sent to the owning supervisor process.

Transitions in different supervisor processes can also be related. Given \((\text{dev}, \text{prop})\) tuple \(C\) is referenced on transition \(A\)’s expression, transition \(B\) may trigger transition \(A\) if \((\text{dev}, \text{prop})\) tuple \(C\) is actuated by one of \(B\)’s actions. In the scenario where there is no influence of the previous relation, i.e. \(B\) and \(C\) are in the same supervisor, but is \(A\) that is placed in a different supervisor, it will also have a cost of one interaction:

- After \(B\)’s action fires and the \(C\) tuple is updated in the same supervisor, one interaction occurs: a publish-subscribe notification is sent from \(C\)’s supervisor to \(A\)’s supervisor, followed by a GET packet sent from \(A\)’s back to \(C\)’s and the corresponding \(A_{\text{GET}}\) packet.

In the case of being \(A\) and \(C\) together in the same supervisor and \(B\) in another, it also has a cost of one:

- After \(B\)’s action fires, one interaction occurs: a NOTIFY packet is sent from \(B\)’s supervisor to \(C\)’s supervisor. Then \(C\)’s value is updated, and \(A\)’s expression is verified in the same supervisor.

The previous facts show us that the base communication cost between a state transition, a combinatorial transition and a \((\text{dev}, \text{prop})\) tuple is the same. If we classify a state machine (i.e. a group of state transitions), a combinatorial transition and a tuple as nodes of an undirected weighted graph, then each edge of the graph represents the number of connections a node establishes with another node. Note the following example, given the following graph connections:

- \((\text{dev}, \text{prop})\) tuple \(P1\) is referenced in \(X\) different state transitions of the same state machine \(M1\) and is referenced by combinatorial transition \(C1\);
- \((\text{dev}, \text{prop})\) tuple \(P2\) is referenced in combinatorial transition \(C2\);
- Combinatorial transition \(C1\) triggers \(W\) different state transitions of the same state machine \(M2\) and triggers 1 state transition of the same state machine \(M1\);
- Combinatorial transition \(C2\) triggers combinatorial transitions \(C1\) and \(C3\) and triggers \(K\) different state transitions of the same state machine \(M2\);
- There are \(Y\) triggering state transitions in \(M1\) that trigger \(Y\) state transitions in \(M2\);

![Figure 29 - Resulting Graph and Maximum Spanning Tree connections in bold](image-url)
We want to get the maximum spanning tree of the graph, in order to obtain the “heaviest” edges that connect all of the nodes. With this tree we are then able to apply single link hierarchical agglomerative clustering to obtain multiple levels of cluster organizations. Iterating through the various levels of the resulting dendrogram, once we find a number of clusters equal to the number of SMs, N, we attribute each cluster to an SM. Then, per cluster, we can drill down into each one of the objects that make up each node (e.g. node is a state machine → 8 state transitions → 8 expressions and 20 actions → 8 AEX commands, …) and produce a specific Assembly file for that SM, resulting in N different Assembly files.

In this example, assuming we had 3 SMs, level 3 of the dendrogram presents 3 clusters where each can be mapped to a different SM, e.g. {M1,P1} to SM1, {M2,C1} to SM2 and {P2,C2,C3} to SM3.

Each cluster is a guarantee that all the involved nodes, which include state machines and (dev, prop) tuples, have the maximum possible number of communications. Then, we are sure that the maximum possible number of nodes were fitted inside each cluster because we chose the level of the dendrogram that had the same number of clusters as the number of SMs. With this solution we maximally benefit from spatial locality within each cluster, minimize the number of interactions between SMs and also address the problem of placing the hardware devices. Since now the (dev, prop) tuples of the instanced hardware devices have an SM assigned, at the generation of the (final) house behavior, a popup appears with the associations of hardware devices to the Supervision Modules, so the user can proceed with the physical connections. In case the user requires that a certain hardware device is connected to a specific SM, he can enforce that on the DomoBus System Specification XML by placing a “supervisor” attribute with the supervisor ID in the corresponding device element. The naïve approach would be to remove that device’s specific (dev, prop) tuple node off the graph and simply place that tuple in the appropriate SM assembly file, therefore future work has room for optimization of this approach.
4.2. DHM Composition

4.2.1. Introduction

The DHM is a Java application tailored for conception of complex automation and operationalization of a DomoBus system. The application loads a DAB database (which currently is a folder in the OS filesystem that has the DAB XML files) and the DomoBus System Specification XML. It has the final objective to generate the commands to be executed by a DomoBus Supervisor process, which will run the logic associated with the set of selected DABs that constitute the final house behavior. The DHM is composed by 7 tabs and a top navigation bar (File, System). The first group of tabs (called the “House” group) is the management part of the DHM application aimed for the non-technical end user. The basic workflow on management tabs is an end user simply associating already existing DABs and to rooms, followed by an instantiation step of their connected hardware devices. Therefore, the end user progressively builds the final house behavior for the loaded system specification and, for intermediate users, they can add notifications on the messages tab and parameters to existing behaviors as well. The second group of tabs (called the “Tools” group) is the DHM set of tools for advanced users (i.e. developers). These are used to create basis and aggregated DABs (explained in section 4.1.1) that can later be used on the management tabs. These tools allow the breakdown of scenarios into combinatorial logic and/or state machines. The Behavior Builder tab has a canvas that shows the updated status of all existing state machines in the DAB and has a drag and drop feature to accelerate development. It also identifies state transition triggers, tracking interactions between state machines and combinatorial logic. Also notice that the Behavior Builder tab also has a sub-tab for adding parameters to DABs. The Aggregator tab is also composed of a large canvas to make the connections between DABs and create intermediate, more complex aggregated DABs.

4.2.2. Top Navbar

The top navbar is composed by two static items, namely File and System, and one dynamic item named Tab that only appears when the Behavior Builder Tab is selected (shown in section 4.2.8). The File item has a subitems named “Open Behaviors Folder” which opens the configured behaviors folder on the host operating system’s file explorer. The System item has three subitems:

- **Load System Specification** – loads a DomoBus System Specification XML file and sets all the house data on the Overview, Rooms, Devices and Messages tabs.
- **Load House Behavior** – loads a DHM behavior XML file of type final. This also forces the loading of the DomoBus System Specification XML file that was used to generate that final DHM behavior. The loading of the DHM behavior will set:
  - The corresponding non-final behaviors to each room on the Rooms tab, along with inter-room connections;
  - Associations of hardware with non-final behaviors’ inputs & outputs on the Devices tab;
  - Messages and their associated transitions and channels on the Messages tab;
  - Parameters and their associated elements on the Parameters tab;
- **Generate House Behavior** – generates a DHM behavior XML file of type *final* and a DHM assembly file. Upon pressing this item, a verification on the instantiation of all inputs and outputs is made. A popup will appear showing all inputs with no hardware device or static value associated with them and will not allow proceeding without fixing all issues. The same popup shows all outputs with no hardware devices associated with them, but it is possible to advance with unconnected outputs, so in this case it is just a warning. Notice that the automated push of the assembly file to the *Supervisor* is not implemented, so the DHM generated file must be manually inputted on the *Supervisor*.

![Popup to fix disconnected inputs and outputs at house behavior generation time](image)

**Figure 32 - Popup to fix disconnected inputs and outputs at house behavior generation time**

### 4.2.3. Overview Tab

The Overview tab is part of the management tabs of the DHM application and offers a summarized description of the house (i.e. system specification) and of the active *final* behavior on the DomoBus system. The top section of the tab has information of the active system specification, such as blueprint of each floor, home address, nº of divisions, nº of devices, nº of supervisors, critical and non-critical services, device classes and external applications. The bottom section has information of the active *final* DHM behavior committed to the system – which at this point we assume it is the one generated by the *Generate House Behavior* item on the top navbar – such as:

- nº of connections – it is the number of connections between inputs and outputs of room DABs plus the number of connections between inputs and outputs with hardware devices;
- nº of child behaviors – it is the number of all *basis* blocks that the loaded *final* block is composed of, since all *aggregated* blocks can be broken down into its constituent *basic* blocks;
- nº of unprotected properties – it is the number of outputs that are connected to hardware devices and are not identified as protected / critical;
- logical depth – it is the maximum number of aggregations a *final* DAB contains inside of itself. In the example below, the logical depth of F1 is 2, because A2 had two aggregation steps and B1 only had one aggregation step;
  - Two *basic* blocks were connected to originate an *aggregated* block A1;
  - A1 was connected with another *basic* block and originated the *aggregated* block A2;
  - Two *basic* blocks were connected to originate an *aggregated* block B1;
  - A2 was connected to B1 and originated the *final* block F1;
- nº of state machine interconnections – it is equal to the number of transition triggers between state machines only.
Critical services are explained in section 4.1.2 and external applications are explained in section 4.1.3.

![Diagram](image)

**Figure 33 - Overview tab of the DHM**

### 4.2.4. Rooms Tab

The Rooms tab is part of the management tabs of the DHM application and allows to set up interconnections between rooms' behaviors and it is composed by N sub tabs, where N is the number of rooms declared on the loaded DomoBus system specification. Each sub tab shows the following:

- A list of all available DHM behaviors that can be loaded as the room’s behavior (*final* behaviors are not eligible) and a Set button;
- Information about the room, such as the floor level, the user access level and the currently active DHM behavior on the room;
- A list of outputs of the current room and a list of inputs of the current room;
- A list of inputs of room R and a list of outputs of room R, where R can be chosen at the “Connections with” scroller next to these lists;
- A “Connect Output Selections” button, a “Connect Input Selections” button and the two respective “Remove Connection” buttons.

When all rooms have a DHM behavior set up, one can start connecting outputs from the list of outputs of the current room to inputs on the list of inputs of room R. Equivalently, one can start connecting outputs from the list of outputs of room R to inputs on the list of inputs of the current room. When making connections, the DHM checks if a connection already exists and only allows one connection to input properties that are not expandable (explained in section 4.1.1).
4.2.5. Devices Tab

The Devices tab allows the association of available hardware (i.e. sensor and actuator devices declared on the loaded DomoBus system specification) with available inputs and outputs of the rooms’ behaviors. The tab is composed by:

- A list of available hardware and an “Associate” button;
- Information about the selected device such as device id, property id, type of device, protection type, location, associated services, user access level and user blocks on the device;
- A textbox to assign a constant value to a selected input and an “Assign” button. These associations will appear as “IX : sysnum_#”, where IX is the prop ID and # is the assigned value;
- An available inputs list, a list of outputs that have room interconnections, a list of outputs that have no room interconnections, a list of existent device associations, a “Remove” button (for existing associations) and a dropdown for choosing a room;
- A “show only above logical depth L” filter scroller that hides inputs and outputs below the selected logical depth L.

Each hardware device can be associated only once to a specific input or output by first selecting the room of interest, then selecting an input or output from one of the lists and then pressing “Associate”. Constant values can also be associated to inputs by using the Assign button and the textbox next to it instead of associating a hardware device. When making associations, the DHM checks if an association already exists and also guarantees that if the hardware device is flagged for critical protection it will only associate to protected outputs (explained in section 4.1.2). By default, the device IDs 0 to 64 are reserved for hardware devices (defined in the DomoBus System Specification XML).
4.2.6. Messages Tab

The Messages tab allows to construct messages that are sent to any custom application – which are typically DomoBus Gateway processes running on an SM that handle the DomoBus Supervision Level protocol – whenever its associated transition is triggered. Currently only static messages are supported (i.e. without addition of \((\text{dev,prop})\) tuple values to the message body). The tab is composed by:

- A dropdown with the list of message types, which currently has only one available value which is “Notification only”;
- A dropdown with the list of all transitions tagged as available for notifications (see section 4.1.3);
- A dropdown with the list of channels (i.e. external applications) of where to send a message;
- Two textboxes (“Event and “Action Taken”) in order to write the message body which consists of what was the event that originated it and the action taken when the transition was triggered;
- A list of the existing messages, a “Remove” button (for existing messages) and an “Add Message” button.

Each message is associated with a transition (tagged as available for notifications) and a channel (i.e. the external application). When the transition is triggered, the message is sent to the respective channel.
4.2.7. Parameters Tab

The Parameters tab allows the parameterization of certain rules' elements from all the room's DHM behaviors. Parameterization in the DHM context is the act of defining the parameter \((dev, prop)\) tuple that controls a parameterizable behavior element that composes a rule. Currently, the parameterizable behavior elements are:

- The test value of an expression condition;
- The delay value of a timing sequence item;

The tab is composed by:

- A list of parameterizable elements and a dropdown with the element types to parameterize which changes the parameterizable elements list content;
- A dropdown to pick the element to parameterize from the selection made at the list of parameterizable elements and the “Add Parameter” button;
- A dropdown to choose the \((dev,prop)\) tuple to be used as a parameter;
- A list of existing parameters and a “Remove” button;
- A checkbox to filter only expressions from state transitions, a scroller to show only elements above a certain logical depth and a dropdown to show only elements from a specific room DAB;

Each parameterized element has either an expression or timing sequence associated with it and also the \((dev,prop)\) tuple whose value is copied to the parameterized element, whenever it changes value.

![Figure 37 - Parameters tab of the DHM](image)

4.2.8. Behavior Builder Tab

The Behavior Builder tab allows the creation of basis DABs, which is the first development phase of a DomoBus house behavior. It is part of the group of tabs called “Tools” and has one of the most important set of features of the DHM. It is responsible for verifying all user actions when creating a basis DAB and guaranteeing that a basis DAB XML file is correctly and efficiently constructed according to DAB XML parsing requirements and format. It can also load an aggregated DAB, but the generated (modified) block will be of type basis. The tab is composed of 4 sub-tabs and a fixed canvas that showcases states and transitions graphically.

The first sub-tab is the “Properties” one, which is composed by the following:
• “New Property” zone, which includes:
  o ID textbox – sets the alphanumeric string ID of the property (required). The prefix “sys_” is reserved by the system and will throw an error if it is contained in the string;
  o Info textbox – sets the description of the property (not required);
  o I/O Type dropdown – sets the role of the property on the block (input, output or virtual);
  o Expansive Expression checkbox and textbox – identifies inputs as expansive and declares their expansive expression (default is unchecked). The structure of the expansive expression is verified against the format enunciated at section 4.1.1.1;
  o Command checkbox – identifies the property as a command (default is unchecked);
  o Initial Value textbox – assigns an initial value to the property (default is 0);
  o Array Size textbox – defines the DomoBus array size of the property. The array size must be an integer between 1 and 255. Required if the property is of type Array;
  o Property Type radio buttons – identifies the type of (dev, prop) tuple (default is 8);
  o State Variable check box and Cardinality checkbox – identifies the output or virtual property as a state variable and chooses its cardinality (default is unchecked and “2”). If this is checked, the property will be shown on the All Variables and State Variables lists on the States sub-tab instead of in the Existing Properties list.

• “Existing Properties” zone, which includes:
  o Existing Properties list – selects an existing property so all of its data is placed on the “New Property” zone and the developer can read its characteristics;
  o Insert button – creates a new property with all data currently set on “New Property” zone;
  o To Expr and To Action buttons, that can be used – when an existing property is selected from the list – to quickly use that variable in a selected expression/action, for its modification.

• “Transitions” zone, which includes:
  o Transitions list – to select an existing transition, of state or combinatorial type, and then proceed with reading its associated expression and actions in the adjacent lists;
    - Add button – adds a transition to the list of existing ones. A popup appears with a textbox for setting the transition alphanumeric string ID (required), another textbox to set the description of the transition (not required) and a checkbox to mark the transition as available for the messaging tab (default is unchecked);
    - Remove button – deletes the selected transition from the DAB.
  o Transition Expression – shows the current expression of the selected transition;
    - Mod/Add button – adds an expression to the selected transition, modify the existing one or add a global scope expression. A popup appears with a textbox to write the expression – composed by a series of conditions and operators – and a dropdown to choose whether to add/modify the transition’s expression or add this expression as global scope. Referencing expressions can be achieved by writing “expr:X” where X is the ID of the expression to be referenced. The possible conditions are =, ≠, <, ≤, >, ≥ and “DO” (where the “DO” condition implies that the list of actions is always executed every time the value of the tuple changes).
o Transition Actions list – shows the actions associated with the selected transition;

- Add button – adds an action to the selected transition or add a global scope action. A popup appears with a dropdown for the type of action (assignment, timing sequence, enable transition, disable transition, or a reference to a global scope action using “act:X” where X is the ID of the action to be referenced, as mentioned in 4.1.1.1), then a group of dynamic dropdowns and, lastly, a dropdown to choose whether to add this action to the selected transition or add it as global scope. The group of dynamic dropdowns become, respectively:

  ▪ Property ID / Value, if the type of action is Assignment. Allowed values for the Value editable dropdown are all numbers or the exact alphanumeric ID of a property (for copying values between properties);

  ▪ Sequence ID / Delay / Action ID, if the type of action is Start Timing Sequence. The Sequence ID dropdown has the IDs of all existing sequences and a “<new>” value. If an existing ID is chosen, the remaining dynamic dropdowns are filled automatically. If the “<new>” value is chosen, then the user needs to fill the dynamic dropdowns. Delay is in seconds and the allowed values for its editable dropdown are all numbers from 1 to 86400. The Action dropdown has the IDs of all global scope sequences and actions;

  ▪ Sequence ID / - / -, if the type of action is Cancel Timing Sequence.

  ▪ Transition ID / - / -, if the type of action is Enable/Disable Transition.

  ▪ Action ID / - / -, if the type of action is a global scope action reference.

- Remove button – deletes the selected action from the DAB.

The second sub-tab is the “Global Scope” one, which includes an actions list, expressions list and timing sequences list, in order to visualize all existing global scope items as well as their IDs by hovering the mouse over a list item. Any item on the lists can be deleted by selecting it and clicking the delete button next to the list. Additionally, the “Transitions” zone shown on the “Properties” sub-tab is also available in this sub-tab to facilitate referencing global scope items on transitions. The third sub-tab is the “States” one, which includes:

  • States list – allows selection of an existing state, which in turn will update the associated state variables and their values on the State Variables list;

    o Insert button – adds a new state to the DAB. A popup will appear with a textbox to insert the name of the state (i.e. its description).

    o Remove button – deletes the selected state from the DAB.

  • State Variables list – shows the state variables associated with selected state and their values.

    o Remove button – removes the selected state variable & its associated value from the list.

  • All Variables list – shows all the state variables that still have unattributed values to states.

    o Var State dropdown – given a selected variable from the All Variables list, shows the possible values, starting on 0 and ascending, depending on the variable’s cardinality;

    o Add to Selected State button – adds selected state variable & Var State to selected state.

  • Info text – shows the current health status of all state machines (i.e. errors).
- **Triggers list** – shows all the unidirectional relations between transitions that trigger a *state* transition. It shows both *combinatorial* $\rightarrow$ *state* and *state* $\rightarrow$ *state* transition triggers. The transition ID shown on the left of the arrow is the trigger transition and the transition ID on the right is the triggered *state* transition. Selecting an item of this list selects the trigger transition on the list of transitions and simultaneously highlight the graphical representation of the triggered *state* transition on the canvas. This aids identification of such scenarios when building a DAB.

Assume the scenario where we have two states S1 and S2, one state variable V1 with cardinality “2”, and an already existing association of V1 = 0 to S1. This would show the string “V1, state:0” on the “State Variables” list and an empty “All Variables” list when state S1 is selected in the States list. Alternatively, it would show the string “V1” on the “All Variables” list – with value “1” on the “Var State” dropdown – and an empty “State Variables” list when state S2 is selected in the States list.

Still on the “States” sub-tab, there is a scroller that works as a filter to showing only transitions, triggers, states and state variables used above a certain logical depth. Additionally, the “Transitions” zone shown on the “Properties” sub-tab is also available in this sub-tab to facilitate visualizing trigger details.

The fourth sub-tab is the “Parameters” one, which is exactly equal to the Parameters tab, but here it is only aware of the current DAB context being built on the Behavior Builder tab.

Lastly, there is a fixed canvas in the Behavior Builder tab, placed underneath the sub-tab region. It constantly shows the updated status of all existing state machines in the DAB, independent of which sub-tab is being shown. In other words, it creates a graphical representation of states and *state transitions*, which can be selected with the mouse to interactively change the current transition selection or state selection (i.e. in their lists, on the sub-tabs). Conversely, given more than one state exists (and therefore appear on the canvas), the developer can select a transition which is not a *state transition* already (from the transitions list), click on a state and then drag & drop onto the target state, in order to convert it to a *state transition*. This drag and drop feature converts non-state transitions to *state transitions* by automatically adding the appropriate combination of state variables and their values as assignment / start sequence actions of that transition, based on the direction of the arrow. At mouse release, if the operation is valid, a popup will appear asking if it is an assignment or start sequence action. For instance, assume the following scenario: states S1 and S2 with an associated state variable V1 of cardinality “2” (V1 = 0 defines S1 and V1 = 1 defines S2) and transition T1 with an arbitrary expression and no actions associated. The developer selects T1 on the list of transitions, clicks on S1, then drags & drops onto S2. A popup appears and the user choses a new assignment action. V1 = 1, appears on T1, just like a new arrow from S1 to S2 labeled “T1”. Notice that, on these drag & drop operations, the transition’s expression is never automatically assigned/modified, unlike actions. The source / condition that makes a transition’s expression evaluate to true is irrelevant for the maintenance of a consistent flow of states, because transitions whose source state is not the current state are disabled. Therefore, it becomes optional (and manual) to include conditions with properties that are state variables in *state* transitions’ expressions, since these can be triggered from *combinatorial or state* transitions’ actions. It is up to the developer to decide how to build the expressions (shown in section 5).
Once a transition has been dragged and dropped between two states successfully – therefore becoming a *state* transition that has actions on state variables – it cannot be dragged and dropped again between any other states. In order to handle it graphically again, the actions on state variables must be manually deleted. Further actions (on non-state variables) can be added to *state* transitions using the “Add” button for actions. This ensures a consistent and controlled environment when graphically creating state machines. Also, a set of states that have common state variables are perceived as a *state machine*, so each state rectangle will receive a small badge in its lower right corner with the ID of the state machine. Clicking the badge selects all states of that *state machine*, allowing to move all states simultaneously on the canvas or make a complete deletion of all states, state variables and associated transitions’ actions that act on those state variables.

Just like the previously mentioned verifications executed on the drag and drop feature, there is a set of other important verifications the Behavior Builder tab keeps track of as well:

- Using the “Add” button for actions, only allows assigning values to outputs and virtuals that are not state variables. Assignment and Start/Cancel Sequence actions on state variables can only be automatically added via drag & drop of transitions between states. Removal of these actions should be done manually though, as mentioned before;
- Manually added “Enable/Disable Transition” actions can only reference *combinatorial* transitions. Enabling/disabling *State* transitions is automatically done at DAB generation time;
- Dragging and dropping between two different state machines will throw a popup with the error “Source and target states do not have the same state variables”;
- *State* transitions that have triggers are graphically shown with a red color, while the ones that do not are shown in blue;
- States that have no state variables assigned are identified as empty in the Info text;
- States that, at some point, are represented by the same set of state variables and associated values, are identified as overlapping states in the Info text;
- If, at any point, a state is represented by a subset of an existing set of state variables, it is identified as overlapping state in the Info text, as well;
- A distinct state machine is only identified once a unique set of state variables covers all the combinations of values and are respectively associated with states;
- When there are empty states or overlapping states (i.e. the states of all state machines are not correctly defined) the drag & drop feature throws a popup with the errors “State X has no state variables assigned” and “States are not unique”, respectively;
- Uniqueness is verified for state names, transition and property alphanumeric IDs;
- Duplication is identified when actions and expressions that already exist in a transition or global scope items are attempted to be re-inserted;
- Faulty expressions and actions (i.e. wrong format) are identified and an error message is shown;
- An expression cannot directly reference an expandable property by its ID but must instead reference the (expandable) expression associated with it (“expr:Y, where Y is the ID of the expandable expression), otherwise an error with be shown;
• In case a property that is a command is referenced in an expression with other conditions, a warning popup will appear showing: “A Command is being referenced in this expression. The existing expression will be completely altered to just ‘do:<ID-of-property>’".

On the top navbar, a dynamic field appears if the Behavior Builder tab is active, called “Tab” which has three options. The “Clear All” option clears all context on the tab. “Load Behavior” clears all context on the tab and then loads a new basis or aggregated DAB for modification. “Generate Behavior” will verify if all state machines are completely defined and if there are missing expressions on any transition. If so, you’ll be prompted to fix those issues manually via a popup.

4.2.9. Aggregator Tab

The Aggregator tab allows the creation of relations between DABs, which is the second development phase of the DomoBus house behavior. It is part of the group of tabs called “Tools” and has an important set of block aggregation features of the DHM. It is responsible for verifying all user actions when aggregating basis and other aggregated DABs and guaranteeing that an aggregated DAB XML file is correctly and efficiently constructed according to DAB XML parsing requirements and format. Notice that “behavior” and “DAB” are used interchangeably. The tab is composed by:

• A list of available behaviors (i.e. DABs from the database) to be inserted on the canvas;
• A list of origin behaviors that compose a selected behavior on the list of available behaviors;
• “Add Behavior” button. It adds the DAB selected on the list of available behaviors to the canvas;
• “Remove Behavior” button. It removes the DAB selected on the canvas from it;
• “Pull Origins” button. It adds all the origin DABs of the selected DAB to the canvas.
• A zone composed by a “Name” textbox, “Description” textbox and “New Behavior” button, which allows to, define the name and description of the resulting DAB of all canvas connections.

Figure 38 - Behavior Builder tab, Properties sub-tab of the DHM
- A canvas which allows for graphical representation of DABs as rectangles, inputs and outputs as circles attached to rectangles, and connections between inputs & outputs of different DABs.

There are a few important verifications the Aggregator tab performs while connecting DABs:

- Inputs that are not expansive can only have one connection from an output;
- An output must always be the source of a connection and an input must always be the target;
- An output and an input must represent a \((dev, prop)\) tuple of the same type to allow a connection;
- Only basis and aggregated DABs can be loaded onto the canvas;
- Can only pull origin DABs from aggregated DABs. The DHM uses the “hash” elements of the aggregated DAB XML file to find the origin DABs and the “hashconn” elements to then recreate the connections between the origin DABs in the canvas.

![Figure 39 - Aggregator tab of the DHM](image)

### 4.3. DHM Implementation

#### 4.3.1. Software Architecture

The DHM was developed using Java and uses the Swing library for GUI programming. A code separation technique was employed to create clear interfaces between front-end and back-end logic even though it followed no specific known coding pattern or framework. Three Java packages were created for separation of concerns in DHM development. This allowed to encapsulate data, provide access control and prevent naming conflicts. The objective was to create an extendable and reproducible way for back-end objects (e.g. DABs), to be represented graphically using front-end objects (e.g. JFrames).

The first package is called `hbGUI` and is used to define the front-end classes and interfaces of the application. The second one is called `hbGen` and is used to define back-end classes that implement logic related with DHM usage, DAB generation and aggregation, DAB parsing, assembly file generation, DomoBus System Specification parsing and its relationship with DHM House tabs. The third one is called `hbPrimitives` and is used to define all classes that constitute a DAB and a House. The main aspects of the DHM software will be summarized next, explaining the usage of the packages.
GUI Structure

The DHM GUI is set up by using classes & interfaces of the hbGUI package. Its structure is the following:

- An hbGUI.GUIApp singleton class extends java.swing.JFrame, is the basis for all Swing objects;
- A java.swing.JMenuBar object (declared on the hbGUI.GUIApp object) has all top navbar items;
- A single java.swing.JTabbedPane object, leftTabPane, that holds all tabs on the left of the hbGUI.GUIApp object. The following classes extend java.swing.JFrame, implement the hbGUI.IUpdater interface and have a single object instantiated and associated with leftTabPane:
  - hbGUI.OverviewGUI class, represents the Overview tab;
  - hbGUI.ConnectorGUI class, represents the Rooms tab;
  - hbGUI.InstantiatorGUI class, represents the Devices tab;
  - hbGUI.MessagingGUI class, represents the Messages tab;
  - hbGUI.ParametizerGUI class, represents the Parameters tab;
  - hbGUI.AggregatorGUI class, represents the Aggregator tab;
  - hbGUI.SingleBehaviorGUI class, represents the Behavior Builder tab;
- A single java.swing.JTabbedPane object, singleTabs, that holds all the sub-tabs of the hbGUI.SingleBehaviorGUI. There is a java.swing.JFrame object associated with singleTabs for each sub-tab:
  - PropertiesFrame object, represents the Properties sub-tab;
  - GlobalScopeFrame object, represents the Global Scope sub-tab;
  - StateFrame object, represents the States sub-tab;
  - ParametersFrame object, represents the Parameters sub-tab;
- For each one of the frames on singleTabs, a java.swing.JPanel object was constructed with all textboxes, labels, buttons, etc. and associated to the correspondent frame;
- An hbGUI.GraphPanel class that extends JPanel and has an attribute of class com.mxGraph. That instanced object for that attribute represents the graphical canvas for state machine visualization. The drag & drop feature for transitions is also implemented here. For the Behavior Tab canvas, a single hbGUI.GraphPanel object is instanced and associated to all of singleTabs JFrames and set as the bottom 50% of the frame, so it always appears on SingleBehaviorGUI;
- An hbGUI.GraphicalItem class that describes a node in a com.mxGraph. Among other attributes, the most important ones of hbGUI.GraphicalItem are the com.mxCell and the java.util.LinkedList<hbGUI.IOGraphical>. In the Behavior Builder tab canvas, there is one instanced hbGUI.GraphicalItem per existing state and each one of these has 4 hbGUI.IOGraphical objects for connecting transitions.
- An hbGUI.IOGraphical class that represents a connection point of an hbGUI.GraphicalItem;
- An hbGUI.GUIHelper class that implements helper functions strictly related with manipulation of com.mxGraph related objects. Some examples of function names are: addConnection, removeConnection, placeIO, customGetToolTip, atLeastOneConnection;
- An hbGUI.IHardwareInfo interface implemented by the hbGen.GUIHardwareInfo class. This interface requests that functions such as String getDevice(), String getProperty(), String getDeviceType(), String getServices() are implemented to show hardware data in the GUI.
• An hbGUI.IBehaviorInfo interface implemented by the GUIBehaviorInfo class. This interface requests that functions such as Map<String, Boolean> getInstancedMap(), Map<String, String> getIoTypes(), String getBehaviorInstanceId(), String getBehaviorGenericId(), LinkedList<hbGen.Transition> getTransitions(), are implemented to show DAB data in the GUI.

• A single java.swing.JTabbedPane object, connectorTabs, that holds all the sub-tabs of the ConnectorGUI. Given N rooms described in the DomoBus System Specification XML, there will be N instances of the same java.swing.JFrame, which was constructed with all textboxes, buttons, etc. of a Room sub-tab. Each one of these JFrames is associated with connectorTabs.

The design of all the other graphical objects (e.g. textboxes, dropdowns, lists, buttons) inside each of the previously mentioned JFrames and JPanels was done with the aid of the Apache NetBeans IDE 11.1 which has a graphical builder view for Swing items.

Tab Interactions

Interaction with the GUI is detected as a java.awt.event. Every Swing item, depending on its class, can have certain types of event classes related with it. For example, a java.swing.JButton has a java.awt.event.MouseEvent object associated to a Mouse Click or Mouse Hover event. The different types of events are handled by callback functions (event handlers) whose function name follows the following pattern: <name-of-object-related-to-event><name-of-event>. For example, addTransitionButtonMouseClicked(java.awt.event.MouseEvent evt) is the prototype of an event handler function for the addTransitionButton JButton object and the MouseClicked event. These functions are placed in the class where the object it handles is declared. In order for an event in the active JFrame to have an effect in another JFrame, the previously mentioned classes must implement the hbGen.IUpdater interface. This interface is composed by two functions:

• public void update(int updateMeaning, Object[] workingObject);
• public void callUpdate(int updateMeaning, Object[] workingObject);

Each class has a reference to the array of objects whose class implements hbGen.IUpdater (which are the tabs objects), added at startup. Therefore, each one of these JFrame can call an update to all objects of that array (i.e. registered tabs), sending an integer to identify the update meaning and any array of objects with the updated data. Then, each one of these classes implements the handling of specific updateMeaning integers related to them and uses the workingObject appropriately. This generic approach allows for extensibility in tab relations. Due to the large number of interactions between tabs, here are a few examples of relationships between tabs implemented by update functions:

• When the Final House Behavior is generated by pressing “Generate House Behavior” on the top navbar, OverviewGUI will update all of the active behavior labels;
• When a new behavior is created at the AggregatorGUI or SingleBehaviorGUI, that new DAB XML file is found, parsed and a Behavior object representing it will be added to ConnectorGUI;
• When a behavior is added to a Room tab of ConnectorGUI, all of its transitions tagged to be available on the Messages tab will be added to the MessagingGUI;
• When a DAB is added to a Room of ConnectorGUI, its parameters are put on ParametizerGUI;
Generator Class

The hbGen.Generator class is instanced three times since there are three separate contexts in the DHM: one for the Rooms/Instantiate/Overview/Messaging/Parameters tabs, another for the BehaviorBuilder tab and, finally, one for the Aggregator tab. This class implements all the logic of DAB creation and aggregation (used in distinct tab contexts), guaranteeing uniqueness of DAB elements, making hardware connections to final DABs, among other important logic. The first time a DAB is parsed into the DHM or is created via Behavior Builder or Aggregator, it creates a hbPrimitives.Behavior object which receives an unique generic ID. That Behavior object is added to a list of behavior types, i.e. non-instantiated behaviors but available to be instantiated on DHM tabs. There are three different contexts to instantiate these behaviors from the previously mentioned list of objects: Rooms tab (and remaining House related tabs), Behavior Builder tab and Aggregator tab, each with a different list of instantiated behaviors. If one of these Behavior objects is associated to a Room in the Rooms tab or placed in the Aggregator tab canvas, a copy of the correspondent object is created and added to the list of instantiated behaviors of the respective context. Then, after connections are done between rooms, or between DABs in the aggregator canvas, when the originating DAB is created (either a final DAB or aggregated DAB, respectively), a new instance is added to the list of non-instantiated behaviors with a unique generic ID (and its XML file is generated). In the Behavior Builder tab only one Behavior can be instantiated at any time, either after being loaded or while being created from scratch. Due to the amount of logic and operations the DAB generation and aggregation processes have, a hbGen.GenHelper class was created to a better breakdown functions for hbGen.Generator to use.

Assembler

The assemble function of the hbGen.Assembler class is its most important function, which receives a final hbPrimitives.Behavior object and the directory of DABs. That function writes the final DAB Assembly file composed of Supervision Library commands that program that DAB in a single supervisor process (assembly file division for multiple supervisor issuing was not implemented due to lack of tools, namely the DCOMM library for Supervision Level intercommunication). The behavior directory can be changed, along with other configurations such as reserved range of the device number for hardware devices, in a configuration file loaded at startup. Due to the amount of logic/operations the assembly process has, a hbGen.AssemblerHelper class was created to organize functions used by the hbGen.Assembler class.

DAB and DomoBus System Specification Parsing

There are two types of file parsing. DomoBus System Specification XML parsing and DAB XML parsing. This is done using a javax.xml.parsers.SAXParser object. The SAXParser is an attribute of the hbGen.SystemParser and hbGen.BehaviorParser classes. DomoBus system specification parsing is done using hbGen.SystemParser, whose parse function returns an hbPrimitives.System object. DAB parsing is done using hbGen.BehaviorParser, whose parse function returns an hbPrimitives.Behavior object. Additionally, the hbGen.BehaviorParser class also implements a toXML function which writes an existing hbPrimitives.Behavior object to XML in DAB format.
4.3.2. DAB XML File Structure

The DAB XML file has three possible statuses. The DAB structures for each status will be shown below. They are unrelated to each other to simplify their structures. Next is an example of a simplistic basis DAB, created according to the reference DAB architecture:

```
<xml version="1.0" encoding="utf-8">
<behavior info="DABB1" status="basis">
<!-- Input Group: --- -- -- -- -- -->
<input id="I1" type="bit" command="true" expansive="true" info="Generic trigger">
  <expression id="1" I1="0.5 KVC"/></expression>
</input>
<!-- Virtual Group: --- -- -- -- -- -->
<virtual id="P1" type="8bit" info="Parameter 1" />
<virtual id="P2" type="8bit" info="Parameter 2" />
<virtual id="V1" type="8bit" info="RGB Red Intensity (0 - 127)" />
<!-- Output Group: --- -- -- -- -- -->
<output id="0" type="8bit" info="Generic (dev,prop) (state var)" nstates="2" />
<output id="0" type="8bit" info="Replica of output 0" />
<output id="NO" type="8bit" info="LED (dev,prop) pair (normal)" />
<!-- Global Scope Expressions: --- -- -- -- -- -->
<expression id="2" expr1="1" info="delay in seconds" />
<!-- Global Scope Actions: -- -- -- -- -- -->
<action id="1" V1="127", seq1();
<sequence id="1">
  <item id="1" delay="10" V1="0"></item>
</sequence>
<!-- Global Scope Timing Sequences: -- -- -- -- -- -->
<sequence id="1">...
</sequence>
<!-- Global Scope Transitions: --- -- -- -- -- -->
<translation id="GT1" info=" clears light to red if it is on">
  <expression id="3" expr1="expression1" />
  <action id="1" V1="127"/>
</translation>
<!-- Global Scope (or combinatorial) transition GT1 checks expression id=2 and sets V1 to 127. -- -- -- -- -- -->
<transition id="GT2" info="Copying value of output 0 to its replica Or">
  <expression id="4" expr1="Document expression1" />
  <action id="5" Or="0"/>
</transition>
<!-- States S1 and S2 are defined and their X and Y positions on the Behavior Tab canvas are stored. -- -- -- -- -- -->
<state id="S1" name="S1: Lights Off" xpos="30" ypos="10"/>
<state id="S2" name="S2: Lights On" xpos="340" ypos="40"/>
<!-- States S1 and S2 are defined and their X and Y positions on the Behavior Tab canvas are stored. -- -- -- -- -- -->
<transition id="T1" info="Button was pressed and output was 0">
  <expression id="5" expr1="expression of 0"/>
  <action id="1" enable:T1, disable:T2; action1/>
</transition>
<transition id="T2" info="Button was pressed and output was 1">
  <expression id="6" expr1="expression of 1", enable:T2, disable:T1; action1/>
</transition>
</behavior>
```

- This is a basis DAB
- The “I1” input is a generic trigger, has 8 bits, is an expansive input with expression id=1 associated and is a command.
- The virtual group of properties, in this case, define 8bit parameters’ properties and an RGB red intensity.
- A state variable, 0, with attribute “nstates” defining it has cardinality = 2, O’s replica Or for interconnecting with other blocks if required, and the NO output that has a physical actuation meaning (LED intensity).
- This a very simple example of global scope expressions and actions being referenced on expression id=3 and action id=4.
- Expression id=2 tests if O’s value equals 1.
- Notice action id=1 performs multiple actions inside it (127 assignment to V1 & starting sequence id=1) but only action id=1 can be referenced.
- Global scope timing sequence id=1 has only one item that, upon starting the sequence will execute the action V1=0 after 3 seconds of delay. This sequence could have more sequence items that would be executed serially.
- Global scope (or combinatorial) transition GT1 checks expression id=2 and sets V1 to 127.
- Global scope Transition GT2 copies O’s value to Or whenever O’s value changes.
- States S1 and S2 are defined and their X and Y positions on the Behavior Tab canvas are stored.
- Each state has a “vars” element declaring all the properties & their values that represent that state.
- Each state can have multiple “connection” elements that declare the transitions that make that state switch to another target state.
- A state machine is recognized by the states with the same combination of state variables (but that have different values).
- Transitions that are part of a state machine and therefore actuate over state variables (and also normal properties).
- Notice actions id=5 and id=7 that implement the automatic enable/disable transition feature for state machines.
- Transitions IDs for parameters are automatically generated with the prefix "sys_" and are appended to the parameter property ID.
- When parameter property P1 changes value, action id=8 is triggered.
- Action id=8 modifies the delay of item id=1 of sequence id=1 with P1’s value.

Figure 40 - Example of basis DAB XML
The *aggregated* status adds new XML elements and new attributes on existent ones upon aggregation:

```xml
<xml version="1.0" encoding="utf-8">  
  <behavior info="Aggregate DAB test" status="aggregated">   
  <!-- Input Group -->   
  <input id="I1" type="bit" command="true" expandable="true" info="Trigger" hash="1"/>   
  </input>   
  <!-- Output Group -->   
  <output id="O1" type="bit" command="false" info="Output 1" hash="1"/>   
  </output>   
  </behavior>   
</xml>
```

- This is an aggregated DAB
- The I1 input, sys_V1 virtual and T2 transition have "hash=1" because they originated from DAB #1.
- sys_V1 already has a "dev" and "prop" associated (even though it would also be done in the construction of the final DAB)

The final status also adds new XML elements and new attributes on existent ones upon aggregation:

```xml
<xml version="1.0" encoding="utf-8">  
  <behavior info="Final DAB test" status="final">   
  <!-- Input Group -->   
  <input id="I1" type="bit" command="true" expandable="true" info="Trigger" hash="1"/>   
  </input>   
  <!-- Output Group -->   
  <output id="O1" type="bit" command="false" info="Output 1" hash="1"/>   
  </output>   
  </behavior>   
</xml>
```

- This is a final DAB
- "roombehavior" associates a DAB (i.e. its hash) to a room.
- "hardwareconnection" associates a device to an input (made in Devices tab)
- The Room information section may include multiple "hardwareconnection" and "roombehavior" elements, which aid in loading this final DAB in the DHM in the future.
4.3.3. Assembly File Structure

The Assembly file is the translation of a final DAB into Supervision Library commands. Many of the DAB/DHM concepts do not directly appear here because their purpose is to building DAB logic, facilitate interconnections and provide a way to load DABs back into the DHM. For example, no references to states, state machines or transitions are made in commands (except on comments). These were concepts used specifically to aid in the construction of DABs and were then translated to the appropriate combination of supervision library commands. Below is an example of an Assembly file that includes the various stages of the file and an example of all relevant mechanisms. For the structure of the EXEC and EXEC_V please refer to section 3.5, and for a list of possible action types that can exist on the AAC command (and their EXEC usage), please refer to section 4.1.1.1.

```
shown here
```

```
- Order of execution starts with declaring input / output / virtual groups, using ADP to declare a (dev,prop) tuple and SPV to set its initial value.
- Then global scope timing sequences are declared. Adding sequences (ASQ), adding items to them (ASI) and adding actions to items (AAS).
- Then transition’s expressions are declared. If the transition expression only has one condition, then its simple condition (ASC) was already declared before. If it has multiple conditions, then AAC’s will be added.
- Then all transitions (i.e. state, combinatorial or parameter) are “materialized” by the adding of actions (AAC) to their expressions.
- Notice that global scope actions that are referenced on transitions are simply copied to each of those transition expression’s actions.
- Parameters and variable-to-variable assignments use EXEC_V, while other actions use EXEC.

```

```

```

```

```
```

```
```

```
```

```
```

```
```

Figure 43 - Example of Assembly file

70
5. Results Analysis

In this section, a use case will be presented in order to test and showcase the DHM program, its workflow and its input and output files. The test comprises all steps for the creation of a complete family house automation, with a medium-sized set of hardware devices (not all rooms were included for simplicity).

The first step will be to conceive potentially reusable scenarios, followed by the formulation of room scenarios. Then, a specific reusable scenario will be chosen for demonstration of the Behavior Builder tab (i.e. resulting basis DAB) and a room scenario will also be chosen for demonstration of the Aggregator tab (i.e. resulting aggregated DAB). All conceived scenarios will be translated to DABs using DHM tools but not all room DABs will be shown due to their size. After all basis DABs and aggregated (room) DABs are created, the DomoBus System Specification XML will be created and loaded into the DHM. Then, using the DHM’s House tabs, each room will have its respective DAB associated to it and interconnection of room DABs will be demonstrated. It is then followed by instantiation of hardware devices, associating them to inputs/outputs of room DABs. This will generate a very large final DAB XML file and Assembly file, which will be truncated at the annexes section.

Finally, the results will be qualitatively evaluated according to the objectives of this work.

5.1. Use Case

Outdoor Scenarios

Outdoor Scenario 1 – New Guest Arrives or Leaves

- When presence is detected in the front yard, garden lights are turned on and sprinklers are interrupted. Assume presence timeout of 30 seconds.
- When presence is not detected in the front yard, garden lights are off and sprinklers continue.
Please refer to Figure 76 in annexes section for the originated DAB XML. This scenario was chosen to demonstrate the Behavior Builder tab due to its simplicity:

![Figure 45 - Outdoor Scenario 1 built on DHM Behavior Builder tab, properties sub-tab](image)

![Figure 46 - Outdoor Scenario 1 built on DHM Behavior Builder tab, states sub-tab](image)

We can see all defined properties in Figure 45. In Figure 46, we can see the definition of both states, all transitions, the expression and actions of TPresenceRefresh. Also, in the canvas we see the graphical representation of states and state transitions, as well as the badge that identifies to which state machine a certain state belongs to. In this case, there’s a single state machine, but there could be more represented state machines in the same canvas. In the Triggers list, we can see an entry that shows TPresenceRefresh triggers the state transition TNoPresenceDetected. There’s a single sequence in the Global Scope sub-tab and there are no defined parameters.

**Outdoor Scenario 2 – Sprinkler Activation based on Temperature and Timeout**

- If temperature rises above X °C and timeout A is equal to zero, sprinklers are activated for Y minutes and timeout A is set to W minutes.
- If timeout B is equal to zero and timeout A is equal to zero, sprinklers are activated for Y minutes and timeout B is set to K minutes.
Please refer to Figure 77 in annexes section for the originated DAB XML.

Outdoor Scenario 3 – Car Arrives or Leaves the Driveway

- If the garage door is closed, and receives an RF signal or correct keypad input, the garage door starts opening, and driveway lights turn on.
- When the garage door reaches fully opened state, timeout A is set to X seconds.
- If the garage door is fully opened and receives either an RF signal or correct keypad input or timeout A is equal to zero, the garage door starts closing and timeout A countdown is canceled.
- When the garage door reaches closed state, driveway lights are turned off.

Please refer to Figure 78 in annexes section for the originated DAB XML.

“Room” Scenario – Outdoor

The room aggregated DAB includes each of the basis DABs in parallel. There’s a relation between scenario 1 & 2 DABs, where SprinklersInterrupt output connects to TurnSprinklersOff input, respectively, Figure 77 for scenario 2 DAB and Figure 78 for scenario 3 DAB.

Living Room Scenarios

Figure 47 - Family house case study - Living Room view
Living Room Scenario 1 – Arriving Home

If no presence was detected in any room of the house for X seconds and the front door unlocked, then:

- Activate all air conditioners of the house to Y ºC;
- Set living room light mode to “watching tv”;
- Turn on living room TV in channel W;
- Open all windows blinds of the house;
- Request Amazon Alexa to speak the remaining daily To Do list tasks;

Please refer to Figure 70 in annexes section for the merged DAB XML of scenarios 1 and 2.

Living Room Scenario 2 – Leaving Home

When the front door gets locked, if after X seconds no presence was detected, then:

- Deactivate all air conditioners of the house;
- Turn off all interior lights of the house;
- Turn off living room TV;
- Close all windows blinds of the house;

Living Room Scenario 3 – Open/Close Living Room Window

- If the living room windows’ blinds are closed, open them until their sensor reads 100% opened.
  If they are opened, close them until their sensor reads 0% opened.

Please refer to Figure 71 in annexes section for the originated DAB XML.

Living Room Scenario 4 – Set Living Room Light Mode

- If a light switch is toggled to position 1 (dinning light mode), then the tv ceiling light is set to X% intensity and the dinning ceiling light is set to Y% intensity.
- If a light switch is toggled to position 2 (tv light mode), then the tv ceiling light is set to W% intensity and the dinning ceiling light is set to K% intensity.
- If a light switch is toggled to position 0 (lights off), then all lights are set to 0% intensity.

Please refer to Figure 72 in annexes section for the originated DAB XML.

Living Room Scenario 5 – Periodic Central Vacuum Cleaning Activation

- Every X minutes activate central vacuum cleaning for Y minutes.

Please refer to Figure 73 in annexes section for the originated DAB XML.

Living Room Scenario 6 – Emergency Shutoff (light, water and gas)

- If any associated button is pressed or fire is detected, shutoff electricity, water and gas.

Please refer to Figure 74 in annexes section for the originated DAB XML.
Room Scenario – Living Room

Scenarios 1 and 2 were merged in the same basis DAB. Besides parallel inclusion of all basis DABs, the aggregated room DAB includes interaction of scenario 1 & 2 DAB with scenario 3 and 4 DABs. WindowBlindsState output will connect to WindowBlindsOpenClose input and LightMode output will connect to 3WayToggleSwitch input. Please refer to Figure 75 in annexes for the originated aggregated DAB XML. This room has been chosen to demonstrate the Aggregator tab that originated this DAB:

![Figure 48 - Living Room Scenario built on DHM Aggregator tab](image)

Kitchen Scenarios

![Figure 49 - Family house case study - Kitchen view](image)
Kitchen Scenario 1 – Automatic Coffee Brewing

If an Amazon Alexa request is received and there is no cup in the coffee brewer, the coffee brewer starts. Please refer to Figure 65 in annexes section for the originated DAB XML.

Kitchen Scenario 2 – Expiration Date Detected on Refrigerator

Daily receive a value with the nearest expiration date on a refrigerator food. If that expiration date is less or equal than X days away, a notification is sent to the user’s mobile phone. Please refer to Figure 66 in annexes section for the originated DAB XML.

Kitchen Scenario 3 – Dishwasher Detected Low Cleaning Supply / Dirty Filter

If a low amount of cleaning supply is detected, a notification is sent to the user’s mobile phone. If the filter is considered dirty, a notification is sent to the user’s mobile phone. Please refer to Figure 67 in annexes section for the originated DAB XML.

Kitchen Scenario 4 – Washing Machine Detected enough Clothes

If the washing machine detects its maximum volume is reached, it starts cycle Y and sends a notification to the user’s mobile phone. Please refer to Figure 68 in annexes section for the originated DAB XML.

Kitchen Scenario 5 – Fire Detection

If the CO2 sensor detects fire then start the sound alarm, open all windows’ blinds of the house and send a notification to the user’s mobile phone. Please refer to Figure 69 in annexes section for the originated DAB XML.

Room Scenario – Kitchen

Each one of the Kitchen individual scenarios originated a single basis DAB. These are all aggregated in parallel, without any interconnections between them, due to their focus on notifications and quick actions. This means the aggregated DAB will simply be a copy of all basis ones, only with unique IDs.

DomoBus System Specification

This system specification file defines device and property types, RBAC, use case’s physical structure and all instanced devices. To restrict the number of possible scenarios, fewer devices were instanced when compared to the ones shown on the room’s figures. Also, several normal and critical services were created and certain hardware was categorized as critical. This allowed to make, for example, devices of the “Locks” and “Alarms” service to only connect protected outputs that were purely governed by state machines. Multiple scalar types were created and attributed to properties according to their purpose. Please refer to Figure 80 in the annexes section for the complete system specification XML.
Room Interconnections

The end user is able to choose, from a database of (room) DABs and their descriptions, which one to associate to each room. He can then create any interconnections between rooms’ inputs and outputs.

In order to demonstrate this, these interactions between the Living Room and the Kitchen were found:

- Kitchen’s CO2 sensor output connects to Living Room’s emergency shutoff logic as ShutoffButton input;
- Living Room’s Alexa output encoded (integer) signal connects to Kitchen’s BrewCoffee input;

![Figure 50 – Kitchen and Living Room Interconnections](image)

The connections can be seen in the “From this room’s Outputs” and “To this room’s Inputs” lists entries with the folder icon.

Instantiation of Hardware Devices

The end user is then able to associate hardware devices to inputs and outputs of the room DABs. In this example, after instantiation messages were created in the Messages tab and then the final house DAB was generated. Please refer to Figure 79 in the annexes section for a truncated version of the final DAB.

The number of conceived scenarios and associated devices was kept to a minimum in order for this use case to be demonstrable. Scenarios can be continuously added as new DABs, then aggregated on each room DAB, reloaded into the Rooms tab (with more interconnections).
5.2. Qualitative Evaluation

Regarding the usage of the DomoBus House Manager to fulfill the use case:

- As an end user, it was seamless to deploy complex automations while being abstracted of the underlying behavior. The manual workflow on the DHM for the end user only required 5 decision stages: 1) loading the correct DomoBus system specification, 2) loading each of the room behaviors into each appropriate room, 3) interconnecting outputs to inputs between rooms, 4) creating messages and 5) associating hardware devices to inputs and outputs of rooms. The workflow took about 7 minutes to be completed and minimized the number of user interactions with behaviors by keeping interaction at the inputs/outputs level and room behavior description only. Stage 2 offered an off-the-shelf experience by allowing to choose room behaviors from a DAB database, based on their scenario descriptions. Stage 3 was particularly flexible, enabling interconnection of behaviors between the living room and kitchen, with potential to create many more connections with other devices, fulfilling the objective of complex scenarios for super-automated homes.

- As an end user, it was easy to connect any device to the same automations while being abstracted of heterogeneous control network architectures and protocols. Stages 1 and 5 offered these abstractions, leveraging on the DomoBus gateway processes on the SM and \((\text{dev}, \text{prop})\) tuples that identify each hardware device. Stage 5 was particularly feature rich, allowing to attribute constant values to inputs, providing granular information on each hardware device,
associations and available inputs & outputs, and also guaranteeing property type and protection consistency.

- As a technical user, it was easy to create automations from scratch using the behavior builder and aggregator tools. The behavior builder tool allowed breakdown of envisioned scenarios into combinatorial logic and/or state machines. Its canvas constantly showed the updated status of all existing state machines in the DAB and the drag and drop feature accelerated its development. The identification of state transition triggers was crucial to manage hard to track interactions between state machines and combinatorial logic. On the other hand, following the basis DAB reference architecture also helped make all produced DABs connection-ready. The DAB definition and its fusion & union properties made sure all deterministic finite automata interactions within an aggregated DAB were self-contained and that connections made in the aggregator tab between DABs were predictable and didn’t disrupt any of the independently functioning state machines. Usage of the aggregator tab was fast and seamless, taking less then a minute to manually obtain a more complex and working aggregated DAB, either for room usage or for further aggregation.

6. Conclusion

Nowadays, software for the creation of behaviors for super automated homes is still limited in the sense that they are not conceived to automate potentially hundreds of devices with better intelligence than simple cause and effect. Due to this problem, the DomoBus system and its device and supervision models were used as flexible and generic resources to conceive the DomoBus Automation Block (DAB) and build the DomoBus House Manager (DHM) program. The DAB definition makes house behaviors self-contained and connectable. The DHM program includes behavior builder tools that ease the construction of complex aggregated logic, generating reusable logic in the form of a DAB. It also provides management tabs for interconnecting room behaviors. Moreover, it abstracts users from the integration details of different device manufacturers with hardware instantiation management. At the same time, it facilitates the continuous creation and management of rules with final DAB loading. Finally, the DHM produces a single assembly file for SM programming but can be extended in the future with the behavior distribution algorithm presented in this work. The DHM becomes a mandatory piece of the DomoBus project for testing real life scenarios in future implementations of a distributed DomoBus system. These are the immediate topics for future work:

- A strategy for deploying assembly files in a distributed DomoBus supervision level was successfully conceived but since no DCOMM library was available at this work’s execution time it was not evaluated. It should be implemented and tested in future releases of the DHM.
- From the DHM perspective, the DomoBus supervisor process has some important limitations that limited the amount of DHM features, such as implementing basic arithmetic operations (+, -, x, ÷) and implementing the possibility to do a V1<I1 (i.e. test between two variables) using the supervision library. These should be addressed in future releases of the supervisor process.
References


Annexes

Ontological specification for a Muzzley Device:

a. Channel
b. Components (see annex 1 for JSON example)
   - id (unique identifier of the component)
   - label (description of the component)
   - classes (JSON array containing a set of classes that describe the component)
c. Properties (see annex 1 for JSON example)
   (General Configuration related)
   - id (unique identifier of the property)
   - label (description of the property)
   - classes (JSON array containing a set of classes that describe the property)
   - schema (URL to the data schema for the property, see annex 1)
   - schema extension (JSON structure which will override the URL schema)
   - __triggerable (boolean that indicates a trigger of Muzzley's automation engine)
   - __actionable (boolean that indicates an action of Muzzley's automation engine)
   - io (property is either readable, writeable, subscribable, or any combination)
   - components (JSON array listing the components this property belongs to)
   - rate limit (frequency through which the property's state will be queried)
   - on change (indicates if the device should only broadcast changes to this property or if it should always broadcast the current state)
   (Automation System related)
   - triggers (if the property is triggerable)
     - condition (the actual condition of the trigger)
     - predicate label (will appear on the automation interface)
     - label (description of the trigger)
     - inputs label (will appear on the automation interface for the given input)
   - actions (if the property is actionable)
     - label (description of the action)
     - inputs label (will appear on the automation interface for the given input)
   - inputs (used for every trigger and action, stating the inputs that these need)
     - id (unique identifier of the needed value for condition inputs)
     - control interface id (unique identifier of the control interface)
     - path (array mapping interface paths and the sent data field)

Figure 52 – Web view interface editor at the selfcare page
### DCN Packet Format

(DCN – DomoBus Control Network)

<table>
<thead>
<tr>
<th>1 byte</th>
<th>2 bytes</th>
<th>2 bytes</th>
<th>1 byte</th>
<th>1 byte</th>
<th>(TLen+0) bytes</th>
<th>1 byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLen</td>
<td>CDevDest</td>
<td>CDevOrig</td>
<td>SNum</td>
<td>CTR</td>
<td>Data</td>
<td>CRC</td>
</tr>
</tbody>
</table>

- **TLen**: Packet total length (bytes)
- **CDevDest**: Destination
- **CDevOrig**: Origin
- **SNum**: Packet sequence number
- **CRC**: Checksum
- **CTR**: Control field

** control level Device address**

<table>
<thead>
<tr>
<th>NAddr</th>
<th>NApp</th>
<th>ADev</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>3 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

- **NAddr**: Node Address (0-255)
- **NApp**: Node Application (0-7)
- **ADev**: Application Device (0-31)

**Figure 53 - DomoBus Control Network Packet Format [28]**

---

**Figure 54**

A) Example of a Fan Speed capability definition in SmartThings; B) JSON POST request body for sending a command to a SmartThings device with a Switch capability [26]
Figure 55 - Structure of the Data field on the DCN packet for the GET and SET message types [28]

Figure 56 - Structure of the Data field on the DCN packet for the NOTIFY message type [28]

Figure 57 - Structure of the Property Descriptor byte on the DCN and Supervision Level packet Data fields [28]

Figure 58 - Structure of the Data field on the DCN packet for the EXEC message type [28]

Figure 59 – DomoBus Supervision Level Packet Format [28]
Figure 60 – Structure of the Data field on Supervision Level packet for GET message type [28]

Figure 61 – Structure of the Data field on Supervision Level packet for the A_GET message type [28]

Figure 62 – Structure of the Data field on Supervision Level packet for the SET message type [28]

Figure 63 – Structure of the Data field on Supervision Level packet for the NOTIFY message type [28]

Figure 64 – Structure of the Data field on Supervision Level packet for the EXEC message type [28]

<behavior info="Automatic Coffee Brewing" status="basis">
  <input id="CoffeeBrewReady" type="bit" command="true" info="Coffee Brewer is ready"></input>
  <input id="CoffeeBrewRequest" type="bit" command="true" info="Requests to brew coffee"></input>
  <output id="BrewCoffee" type="bit" command="true" info="Brews coffee if machine is ready">
    <transition id="1">CoffeeBrewRequest=1 AND CoffeeBrewReady=1</transition>
    <action id="1">BrewCoffee=1</action>
  </output>
</behavior>

Figure 65 - Kitchen’s Scenario 1 DAB

<behavior info="Food Expiration" status="basis">
  <input id="ExpirationLimitParameter" type="bit" info="In days"></input>
  <input id="RefrigeratorDailyExpirationAlerts" type="bit" command="true" info="Daily value of the nearest expiration date in days"></input>
  <transition id="ExpirationAlert" info="[H] Alerts there are expiring items in the refrigerator">
    <expression id="1">RefrigeratorDailyExpirationAlerts=1</expression>
  </transition>
</behavior>

Figure 66 - Kitchen’s Scenario 2 DAB
Figure 67 - Kitchen's Scenario 3 DAB

Figure 68 - Kitchen's Scenario 4 DAB

Figure 69 - Kitchen's Scenario 5 DAB
87

Figure 70 - Living Room’s Scenario 1 and 2 DAB
Figure 71 - Living Room’s Scenario 3 DAB

```
<behavior info="Open and close window blinds" status="basis">
  <input id="WindowBlindsPercentRead" type="bit" info="Reads the opened percentage"/>
  <input id="WindowBlindsOpenClose" type="bit" info="Sets desired blinds state"/>
  <output id="WindowBlindsServo" type="bit" info="Actuate servo to open blinds"/>
  <state id="51" name="Blinds Closed" xpos="160.0" ypos="86.0">
    <var id="BlindsStateVar">0</var>
    <connection to="52" transition="TBlindsOpening"></connection>
  </state>
  <state id="52" name="Blinds Opening" xpos="160.0" ypos="86.0">
    <var id="BlindsStateVar">1</var>
    <connection to="53" transition="TBlindsOpen"></connection>
  </state>
  <state id="53" name="Blinds Open" xpos="160.0" ypos="140.0">
    <var id="BlindsStateVar">2</var>
    <connection to="54" transition="TBlindsClosing"></connection>
  </state>
  <state id="54" name="Blinds Closing" xpos="160.0" ypos="140.0">
    <var id="BlindsStateVar">3</var>
    <connection to="55" transition="TBlindsClosed"></connection>
  </state>
  <transition id="TBlindsOpen" info="Blinds open">
    <?xml version="1.0" encoding="ISO-8859-1"?>
    <action id="55">WindowBlindsOpenClose=1 AND WindowBlindsPercentRead=0 AND BlindsStateVar=0</action>
    <action id="56">disableTBlindsOpening, enableTBlindsOpen, disableTBlindsClosing, disableTBlindsClosed</action>
  </transition>
  <transition id="TBlindsClosing" info="Blinds closing">
    <expression id="57">WindowBlindsOpenClose=1 AND WindowBlindsPercentRead=100 AND BlindsStateVar=2</expression>
    <action id="58">disableTBlindsOpening, disableTBlindsOpen, enableTBlindsClosing, disableTBlindsClosed</action>
  </transition>
  <transition id="TBlindsClosed" info="Blinds closed">
    <expression id="59">WindowBlindsOpenClose=1 AND WindowBlindsPercentRead=0 AND BlindsStateVar=3</expression>
    <action id="60">disableTBlindsOpening, disableTBlindsOpen, disableTBlindsClosing, enableTBlindsClosed</action>
  </transition>
</behavior>
```

Figure 72 - Living Room’s Scenario 4 DAB

```
<behavior info="Two zones, three Light modes" status="basis">
  <input id="ModeZone1Percent" type="bit" info="Sets mode 1 zone 1 percent"/>
  <input id="ModeZone2Percent" type="bit" info="Sets mode 1 zone 2 percent"/>
  <input id="ModeZone1Percent" type="bit" info="Sets mode 2 zone 1 percent"/>
  <input id="ModeZone2Percent" type="bit" info="Sets mode 2 zone 2 percent"/>
  <input id="3WayTogglerSwitch" type="bit" info="To set the mode"/>
  <output id="LightsZone1" type="bit" info="Zone 1 Lights intensity percent"/>
  <output id="LightsZone2" type="bit" info="Zone 2 Lights intensity percent"/>
  <transition id="TMode0" info="Mode 0">
    <expression id="61">3WayTogglerSwitch=0</expression>
    <action id="62">LightsZone1=0, LightsZone2=0</action>
  </transition>
  <transition id="TMode1" info="Mode 1">
    <expression id="63">3WayTogglerSwitch=1</expression>
    <action id="64">LightsZone1=ModeZone1Percent, LightsZone2=ModeZone2Percent</action>
  </transition>
  <transition id="TMode2" info="Mode 2">
    <expression id="65">3WayTogglerSwitch=2</expression>
    <action id="66">LightsZone1=ModeZone1Percent, LightsZone2=ModeZone2Percent</action>
  </transition>
</behavior>
```
Figure 73 - Living Room's Scenario 5 DAB

<behavior info="Periodically trigger central vacuum cleaning" status="basis">
  <input id="CleaningTimeout" type="8bit" info="In minutes"></input>
  <input id="CleaningDuration" type="16bit" info="In minutes"></input>
  <output id="CentralVacuum" type="8bit" info="Activates central vacuum cleaner"></output>
  <sequence id="1">
    <item id="1" delay="5">CentralVacuum=0</item>
  </sequence>
  <sequence id="2">
    <item id="2" delay="1440">CleaningTimeout==0</item>
  </sequence>
  <transition id="Tclean" info="Initiates vacuum cleaning">
    <expression id="1">CleaningTimeout==0</expression>
    <action id="1">CentralVacuum=1, CleaningTimeout=1440</action>
  </transition>
  <transition id="TResetShutOff" info="Resets shutoff of Electricity, Water and Gas">
    <expression id="2">GasShutoff=0, WaterShutoff=0, ElectricityShutoff=0</expression>
    <action id="2">GasShutoff=1, WaterShutoff=1, ElectricityShutoff=1</action>
  </transition>
</behavior>

Figure 74 - Living Room's Scenario 6 DAB

<behavior info="Trigger emergency shutoff" status="basis">
  <input id="FireDetection" type="8bit" info="Detects fire"></input>
  <input id="ShutoffButton" type="8bit" command="true" expandable="true" info="Signal to shutoff resources">
    <expression id="1">ShutoffButton=1 OR 0x</expression>
  </input>
  <input id="ResetShutOff" command="true" type="8bit" info="Resets shutoff"></input>
  <output id="ElectricityShutoff" type="8bit" info="Cuts electricity off"></output>
  <output id="WaterShutoff" type="8bit" info="Cuts water supply off"></output>
  <output id="GasShutoff" type="8bit" info="Cuts gas supply off"></output>
  <transition id="Shutoff" info="Shutoff of Electricity, Water and Gas">
    <expression id="2">FireDetection=1 OR expr1</expression>
    <action id="2">GasShutoff=1, WaterShutoff=1, ElectricityShutoff=1</action>
  </transition>
  <transition id="TResetShutOff" info="Resets shutoff of Electricity, Water and Gas">
    <expression id="3">ResetShutOff==0</expression>
    <action id="3">GasShutoff=0, WaterShutoff=0, ElectricityShutoff=0</action>
  </transition>
</behavior>
Figure 75 - Living Room aggregated DAB
Figure 76 - Outdoor Scenario 1 DAB
Figure 77 - Outdoor Scenario 2 DAB
Figure 78 - Outdoor Scenario 3 DAB
Figure 79 – Truncated Final DAB (removed room logic)
<DeviceType ID="9" Name="Sprinkler">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="5" Name="Contact Sensor">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="7" Name="HVAC">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
        <Property ID="2" Name="Temp" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="8" Name="TV">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
        <Property ID="2" Name="Channel" AccessMode="EN" ValueType="SCALAR" RefValueType="4"/>  
        <Property ID="3" Name="SoundLevel" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="9" Name="CO2 Sensor">  
    <PropertyList>  
        <Property ID="1" Name="Alarm" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="10" Name="Ultrasonic Sensor">  
    <PropertyList>  
        <Property ID="1" Name="Distance" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="11" Name="Two-way Switch">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="12" Name="Vacuum">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="13" Name="Motor">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="14" Name="Shutter">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="15" Name="Household Appliance">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
        <Property ID="2" Name="Signal" AccessMode="EN" ValueType="SCALAR" RefValueType="1"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="16" Name="Radiofrequency">  
    <PropertyList>  
        <Property ID="1" Name="On/Off" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="17" Name="Wirepad">  
    <PropertyList>  
        <Property ID="1" Name="Signal" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="18" Name="Temperature Sensor">  
    <PropertyList>  
        <Property ID="1" Name="Temp" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
    </PropertyList>  
</DeviceType>

<DeviceType ID="19" Name="C" AccessMode="EN" ValueType="SCALAR" RefValueType="3"/>  
</DeviceTypeList>
<DeviceList>
  <Device ID="1" RefDeviceType="2" Name="Living Room Motion Sensor" Address="#0100" RefDevice="1" />
  <DeviceServiceList>
    <DeviceService RefService="1" />
  </DeviceServiceList>
</Device>

<Device ID="2" RefDeviceType="6" Name="Front Door Lock Sensor" Address="#0100" RefDevice="2" />
  <DeviceServiceList>
    <DeviceService RefService="3" />
  </DeviceServiceList>
</Device>

<Device ID="3" RefDeviceType="7" Name="Living Room Air Cond." Address="#0100" RefDevice="3" />
  <DeviceServiceList>
    <DeviceService RefService="4" />
  </DeviceServiceList>
</Device>

<Device ID="4" RefDeviceType="2" Name="Living Room TV" Address="#0100" RefDevice="2" />
  <DeviceServiceList>
    <DeviceService RefService="5" />
  </DeviceServiceList>
</Device>

<Device ID="5" RefDeviceType="11" Name="Living Room Window Switch" Address="#0100" RefDevice="5" />
  <DeviceServiceList>
    <DeviceService RefService="5" />
  </DeviceServiceList>
</Device>

<Device ID="6" RefDeviceType="6" Name="Front Door Lock" Address="#0100" RefDevice="6" />
  <DeviceServiceList>
    <DeviceService RefService="3" />
  </DeviceServiceList>
</Device>

<Device ID="7" RefDeviceType="10" Name="Living Room Windows Sensor" Address="#0100" RefDevice="7" />
  <DeviceServiceList>
    <DeviceService RefService="3" />
  </DeviceServiceList>
</Device>

<Device ID="8" RefDeviceType="8" Name="Living Room Alexa" Address="#0100" RefDevice="8" />
  <DeviceServiceList>
    <DeviceService RefService="5" />
  </DeviceServiceList>
</Device>

<Device ID="9" RefDeviceType="9" Name="Living Room CO2" Address="#0100" RefDevice="9" />
  <DeviceServiceList>
    <DeviceService RefService="6" />
  </DeviceServiceList>
</Device>

<Device ID="10" RefDeviceType="12" Name="Living Room Vacuum" Address="#0100" RefDevice="10" />
  <DeviceServiceList>
    <DeviceService RefService="7" />
  </DeviceServiceList>
</Device>

<Device ID="11" RefDeviceType="13" Name="Living Room Window Motor" Address="#0100" RefDevice="11" />
  <DeviceServiceList>
    <DeviceService RefService="3" />
  </DeviceServiceList>
</Device>

<Device ID="12" RefDeviceType="3" Name="Living Room Ceiling Light 1" Address="#0100" RefDevice="12" />
  <DeviceServiceList>
    <DeviceService RefService="2" />
  </DeviceServiceList>
</Device>

<Device ID="13" RefDeviceType="3" Name="Living Room Ceiling Light 2" Address="#0100" RefDevice="13" />
  <DeviceServiceList>
    <DeviceService RefService="2" />
  </DeviceServiceList>
</Device>

<Device ID="14" RefDeviceType="14" Name="Electricity Shutoff" Address="#0100" RefDevice="14" />
  <DeviceServiceList>
    <DeviceService RefService="5" />
  </DeviceServiceList>
</Device>
</DeviceList>
<Device ID="10" RefDeviceType="14" Name="Water Shutoff" Address="#0100" RefDivision="2"
<DeviceServiceList>
  <DeviceService RefService="8" />
</DeviceServiceList>
</Device>

<Device ID="16" RefDeviceType="14" Name="Gas Shutoff" Address="#0100" RefDivision="3"
<DeviceServiceList>
  <DeviceService RefService="9" />
</DeviceServiceList>
</Device>

<Device ID="17" RefDeviceType="15" Name="Refrigerator" Address="#0100" RefDivision="3"
<DeviceServiceList>
  <DeviceService RefService="9" />
</DeviceServiceList>
</Device>

<Device ID="18" RefDeviceType="15" Name="Coffee Brewer" Address="#0100" RefDivision="3"
<DeviceServiceList>
  <DeviceService RefService="9" />
</DeviceServiceList>
</Device>

<Device ID="19" RefDeviceType="15" Name="Dishwasher" Address="#0100" RefDivision="3" S
<DeviceServiceList>
  <DeviceService RefService="9" />
</DeviceServiceList>
</Device>

<Device ID="20" RefDeviceType="15" Name="Washing Machine" Address="#0100" RefDivision="3"
<DeviceServiceList>
  <DeviceService RefService="9" />
</DeviceServiceList>
</Device>

<Device ID="21" RefDeviceType="15" Name="Kitchen CO2" Address="#0100" RefDivision="3" S
<DeviceServiceList>
  <DeviceService RefService="9" />
</DeviceServiceList>
</Device>

<Device ID="22" RefDeviceType="16" Name="Garage Door RF Reader" Address="#0100" RefDiv
<DeviceServiceList>
  <DeviceService RefService="8" />
</DeviceServiceList>
</Device>

<Device ID="23" RefDeviceType="17" Name="Driveway Light 1" Address="#0100" RefDivi
<DeviceServiceList>
  <DeviceService RefService="2" />
</DeviceServiceList>
</Device>

<Device ID="24" RefDeviceType="17" Name="Driveway Light 2" Address="#0100" RefDivi
<DeviceServiceList>
  <DeviceService RefService="2" />
</DeviceServiceList>
</Device>

<Device ID="25" RefDeviceType="17" Name="Driveway Light 3" Address="#0100" RefDivi
<DeviceServiceList>
  <DeviceService RefService="2" />
</DeviceServiceList>
</Device>

<Device ID="26" RefDeviceType="17" Name="Driveway Light 4" Address="#0100" RefDivi
<DeviceServiceList>
  <DeviceService RefService="2" />
</DeviceServiceList>
</Device>

<Device ID="27" RefDeviceType="17" Name="Driveway Light 5" Address="#0100" RefDivi
<DeviceServiceList>
  <DeviceService RefService="2" />
</DeviceServiceList>
</Device>

<Device ID="28" RefDeviceType="17" Name="Driveway Light 6" Address="#0100" RefDivi
<DeviceServiceList>
  <DeviceService RefService="2" />
</DeviceServiceList>
</Device>

<Device ID="29" RefDeviceType="17" Name="Garden Light 1" Address="#0100" RefDivision="1"
<DeviceServiceList>
  <DeviceService RefService="2" />
</DeviceServiceList>
</Device>
<Device ID="30" RefDeviceType="3" Name="Garage Light 2" Address="#0100" RefDivision="1" />
</Device>
<DeviceID="31" RefDeviceType="3" Name="Garage Light 3" Address="#0100" RefDivision="1" />
</Device>
<Device ID="32" RefDeviceType="6" Name="Sprinkler 1" Address="#0100" RefDivision="1" S />
</Device>
<Device ID="33" RefDeviceType="6" Name="Sprinkler 2" Address="#0100" RefDivision="1" S />
</Device>
<Device ID="34" RefDeviceType="6" Name="Sprinkler 3" Address="#0100" RefDivision="1" S />
</Device>
<Device ID="35" RefDeviceType="12" Name="Front Yard Motion Detector" Address="#0100" Re />
</Device>
<Device ID="36" RefDeviceType="10" Name="Garage Door Sensor" Address="#0100" RefDivision="1" />
</Device>
<Device ID="37" RefDeviceType="13" Name="Garage Door Motor" Address="#0100" RefDivision="1" />
</Device>
<Device ID="38" RefDeviceType="17" Name="Garage Door Keypad" Address="#0100" RefDivision="1" />
</Device>
<Device ID="39" RefDeviceType="18" Name="Outside Temperature Sensor" Address="#0100" B />
</Device>
<Device ID="40" RefDeviceType="1" Name="Shutoff Button" Address="#0100" RefDivision="1" Super: />
</Device>
<Device ID="41" RefDeviceType="1" Name="Reset Shutoff" Address="#0100" RefDivision="1" Super: />
</Device>
</Device>
</DeviceList>
<MessageChannelList>
<Channel ID="19" PropID="128" Type="Notification" ResponseType="N/A" ExternalAppName="Email" />
</MessageChannelList>
</House>
</FloorList>
</FloorID="1" Name="Ground Floor" HeightOrder="0" />
</FloorList>
</DivisionList>
<Division ID="1" Name="Outdoor" RefFloor="#1" AccessLevel="1" />
<Division ID="2" Name="Living-room" RefFloor="#1" AccessLevel="1" />
<Division ID="3" Name="Kitchen" RefFloor="#1" AccessLevel="1" />
</DivisionList>
</House>
Figure 80 - DomoBus System Specification XML of the use case's Family House