João Pedro Gamito Taborda

Licenciado em Engenharia Informática e de Computadores

A Domain Specific Language for Domotic Systems

Dissertação para obtenção do Grau de Mestrado em
Engenharia Informática

Orientador: Vasco Amaral, Prof. Auxiliar, Universidade Nova de Lisboa

Co-orientador: Paulo Carreira, Prof. Auxiliar,Instituto Superior Técnico

Júri:

Presidente: Prof. Doutor Nuno Manuel Ribeiro Preguiça

Arguente: Prof. Doutor Fernando Manuel Pereira da Costa Brito e Abreu

Vogal: Prof. Doutor Vasco Miguel Moreira do Amaral

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2014

A Domain Specific Language for Domotic Systems

Everything is okay in the end, if it's not ok, then it's not the end…

ACKNOWLEDGEMENTS

First and above all, I want to thank my soulmate for all the patience, dedication and support. Without her, it would have been much more difficult to write this document, since it was her perseverance that pushed me into doing more and better. Thank you Cátia for being always there, not only by your motivation, but also for all the opinions and useful comments on crucial moments of the thesis development.

To Prof. Dr. Vasco Amaral, who was a great advisor in several aspects of the work. His knowledge and guidance were essential for keeping the developing alive. I have appreciated very much all his opinions and directions, which significantly contributed for accomplishing the next stage. To Prof. Paulo Carreira, an appreciation note for organizing the meetings conducted at IST.

A very special word to my parents, who supported me through this period and sustained some absence in the more stressful moments.

Finally, to my brother who is always distracting me with his meaningless subjects. Those distractions were (most of the times) a brief moment of evasion from all the work.

RESUMO

Seguindo as tendências de modernização, a necessidade de ter uma casa totalmente automatizada, tem sido um conceito que tem vindo a crescer ao longo dos anos à medida que a tecnologia evolui e as pessoas necessitam de soluções que lhes poupem tempo. Um sistema de automação fornece uma forma de simplificar algumas das rotinas diárias, permitindo-nos ter mais tempo livre para realizar actividades em que somos realmente necessários. Existem neste meio alguns sistemas que se destinam a este fim, mas este tipo de tecnologia ainda se encontra a dar os primeiros passos e portanto ainda está longe de fornecer ao utilizador o tão desejado controlo sobre uma habitação. A razão principal é que este tipo de sistemas não implementa alguns princípios importantes, tais como a adaptabilidade, a extensibilidade e a evolução.

Os sistemas mencionados, que são desenvolvidos usando uma abordagem ascendente, são mais adequados à utilização por parte de programadores e especialistas no domínio, descartando os utilizadores comuns aos quais apenas resta produtos difíceis de controlar ou com interfaces pouco flexíveis. Além disso, comportamentos mais complexos não são considerados, pois são bastante difíceis de implementar devido à necessidade de conjugar prioridades, conflitos e calibração dos dispositivos. Outra das desvantagens, é o facto de estas soluções só poderem ser adquiridas a um elevado custo, existindo ainda a limitação de serem de difícil configuração por pessoas que não são conhecedoras dos termos técnicos envolvidos, quando instalados e em funcionamento.

É por isso necessário criar uma ferramenta que permita sobretudo que o utilizador possa realizar as mais variadas acções no domínio da automação, sem que no entanto estas se tornem excessivamente complicadas. Também é desejável que essa ferramenta seja independente dos dispositivos, para que possa ser reutilizada, ou seja, que siga uma abordagem orientada a modelos (MDD). Uma vez que o domínio de automação tem alguns conceitos muito específicos, a utilização de modelos deverá ser aliada a uma linguagem específica de domínio (DSL), para que utilizadores desta área reconheçam facilmente os conceitos envolvidos. Através destes dois métodos, é possível criar uma solução que se encontra adaptada para o utilizador comum, mas que ao mesmo tempo possa ser utilizada por especialistas e programadores devido às várias camadas de abstracção que diminuem a complexidade de utilização.

O objectivo desta tese é a realização de uma linguagem específica de domínio (DSL) que utilize a abordagem de desenvolvimento por modelos (MDD), com a finalidade de suportar conceitos de automação doméstica. Nesta implementação, o desenvolvimento de cenários simples e complexos será uma das preocupações mais relevantes, sendo que o suporte a outro tipo de funcionalidades, como a possibilidade de agendamento de tarefas, será também tido em conta devido a ser uma funcionalidade limitada nas soluções actuais.

Palavras-chave: Automação doméstica, Domótica, Sistemas automáticos, Cenários de automação, Linguagens específicas de domínio, Desenvolvimento orientado por modelos

ABSTRACT

To cope with modernity, the interesting of having a fully automated house has been increasing over the years, as technology evolves and as our lives become more stressful and overloaded. An automation system provides a way to simplify some daily tasks, allowing us to have more spare time to perform activities where we are really needed. There are some systems in this domain that try to implement these characteristics, but this kind of technology is at its early stages of evolution being that it is still far away of empowering the user with the desired control over a habitation. The reason is that the mentioned systems miss some important features such as adaptability, extension and evolution.

These systems, developed from a bottom-up approach, are often tailored for programmers and domain experts, discarding most of the times the end users that remain with unfinished interfaces or products that they have difficulty to control. Moreover, complex behaviors are avoided, since they are extremely difficult to implement mostly due to the necessity of handling priorities, conflicts and device calibration. Besides, these solutions are only reachable at very high costs, yet they still have the limitation of being difficult to configure by non-technical people once in runtime operation.

As a result, it is necessary to create a tool that allows the execution of several automated actions, with an interface that is easy to use but at the same time supports all the main features of this domain. It is also desirable that this tool is independent of the hardware so it can be reused, thus a Model Driven Development approach (MDD) is the ideal option, as it is a method that follows those principles. Since the automation domain has some very specific concepts, the use of models should be combined with a Domain Specific Language (DSL). With these two methods, it is possible to create a solution that is adapted to the end users, but also to domain experts and programmers due to the several levels of abstraction that can be added to diminish the complexity of use.

The aim of this thesis is to design a Domain Specific Language (DSL) that uses the Model Driven Development approach (MDD), with the purpose of supporting Home Automation (HA) concepts. In this implementation, the development of simple and complex scenarios should be supported and will be one of the most important concerns. This DSL should also support other significant features in this domain, such as the ability to schedule tasks, which is something that is limited in the current existing solutions.

Keywords: Home Automation, Domotic, Automated systems, Automation scenarios, Domain Specific Languages, Model Driven Development

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ACRONYMS AND ABBREVIATIONS

AmI – Ambient Intelligence

DSL – Domain Specific Language

EGL – Epsilon Generation Language

EHS – European Home Systems Protocol

EIB – European Installation Bus

EMF – Eclipse Modeling Framework

EOL – Epsilon Object Language

ETL – Epsilon Transformation Language

EVL – Epsilon Validation Language

FBD – Function Block Diagram

GMF – Graphical Modeling Framework

GPL – General Purpose Language

GUI – Graphical User Interface

HA – Home Automation

IDE – Integrated Development Environment

IL – Instruction List

LD – Ladder Diagram

M2M – Model-to-model

M2T – Model-to-text

MARTE – Modeling and Analysis of Real-Time and Embedded Systems

MDD – Model Driven Development

OCL – Object Constraint Language

OMG – Object Management Group

PLC – Programmable Logic Controllers

RF – Radio Frequency

RTES – Real-Time and Embedded Systems

SD – State diagrams

SFC – Sequential Function Chart

ST – Structured Text

UML – Unified Modeling Language

1

INTRODUCTION

Home Automation (HA) has been lately within the focus of the general society's interest, since the increasing proliferation of technology encourages people to automate several aspects of their daily life [1]. Such automation increases productivity, saves costs and even provides comfort to its users. However, several limitations have been hampering its wide adoption, namely, the cost of devices, lack of interoperability, and above all, the complexity of use and inadequate adaptation to non-technical people.

The perfect automated house, is generally idealized as a system that adapts itself to the users' behavior and, when needed, interacts with them. That house would be composed by several devices that should provide a high degree of comfort, security and energy savings [1][2], since the aim of a HA system is to "improve the quality of life of its inhabitants" [1]. In order to provide the best comfort, the house should create the perfect room atmosphere accordingly to the users' needs; the security is achieved through warnings about devices that are on and could represent a risk; for energy saving, the house would be smart enough to turn off certain devices when they are not required. However, there is a need to make the necessary trade-offs between the inhabitants' desires, the waste of energy and the generation of alerts about energy consumption [3][4]. Regrettably, the present technology cannot yet provide an affordable system like the one mentioned, but a path can be drawn towards it.

In this domain, it is possible to find specific systems that attempt to provide some of the previous characteristics, but they lack the proper coordination between the key elements mentioned above. Moreover, they are built as a product addressed to people that have to be knowledgeable about this field and who understand the intricacies of the technology involved. In fact, those systems have some flaws in terms of complexity and abstraction, because an attempt to create a more complex behavior ends up in an complicated scenario that is not understandable by the majority of non-technical people [5]. These complex behaviors must take into account certain aspects such as priorities, conflicts and device calibration, which makes them complicated to implement. The exclusion of these kind of behaviors has two main reasons behind it: one of them is because they introduce the possibility for certain behaviors to fail in unexpected ways; the other is the hardware-dependency that exists between the software and the devices controlled.

Despite the fact that those systems could be a solution to some people, one of the main problems in HA is the lack of consensus around a standard [1]. This fact leads to the problems that this area currently suffers, because it means that each company creates its own environment to program the devices, and once a user chooses a technology, all the devices that he acquires should be compatible with a specifc protocol created by that company for the comunication between devices.

1.1 MOTIVATION

The development of HA applications is, nowadays, strongly platform dependent. With this in mind, one can be aware that the available solutions do not have the adaptation desired in this field, which is a limiting factor when we need to develop scenarios with great complexity. A possible way to solve this dependency and lack of flexibility is by introducing abstraction layers and promote the reuse of components in a systematic way. The Model Driven Development (MDD) methodology is a possible approach to achieve the desired degree of independency [6]. This method, splits up the specification of a system's functionality from the implementation of that functionality on a specific platform [7]. This means that with MDD, the code is generated from platform-independent models and one does not need to worry about implementation details, such as the platforms supported or the programming techniques used [2], which are properly treated at the different levels of abstraction.

To be effectively used on HA, the MDD approach should be combined with Domain Specific Languages (DSL) that are optimized to handle specific problems in a certain domain. This type of combination provides an abstraction layer that will be used to isolate the system’s design from its implementation details, which allows a separation of concerns that offers more flexibility to the designer of the HA system.

Since there still not exists a complete solution that handles the emergent problems in the HA domain, a DSL is a possible way to define a language that can be flexible enough to define complex behaviors, which are not supported by the technology that currently exists [8]. In HA there are two types of devices, the sensors that are meant to be aware of the environment’s occurrences and the actuators that perform specific actions in response to the sensors’ activation.

A simple behavior could be defined as a situation where there is only a sensor and an actuator, and when the sensor is triggered, the actuator reacts. This type of behavior is currently supported by the systems previously mentioned, as the action has a straightforward execution. The problem lies in complex behavior, that is composed by more devices, and with more devices comes greater complexity.

A complex behavior is a situation where are several devices involved, for example, an actuator and two sensors that are connected to that actuator. A possible scenario is to consider the actuator as a light bulb and the sensors as a presence sensor and as a daylight sensor. The first sensor turns the light on, if there is someone in the room and the other, turns the light on if the light outside is scarce. The question is what happens if it is dark outside and there is no one in the room. The light should be off because there is no presence or should be on because it is night time?

There is no ideal answer to the question above, because it depends on the user’s decision. This means that for a system, the stated problem is a conflict that must be resolved. In this more complex scenario, only two sensors and an actuator are being considered. To support real HA, there are several problems like this to contemplate, since there will be a great number of devices that will have these sort of conflicts. Another concern in this domain, is the possibility of defining scheduled actions, which is a

feature that have to be present in a system that is said to be automated, because the possibility to define a set of actions to run at a pre-defined time is a basic characteristic.

In a HA system, the complexity increases with the number of devices in that system, with the number of relations between them [2] and with the amount of functionalities it offers. To manage the connection between devices and the conflicts at the same time, the DSL choice stands as an effective possibility, since it can represent the elements in a graphical way, which increases the expressiveness and the usability that other solutions do not deliver. With a DSL, it is also possible to built complex behaviors, as its creation depends on the tools and the way the language is defined.

1.2 Objectives and Problem statement

The main goal of this thesis is to design a DSL for the development of HA systems that follows the MDD methodology. This DSL aims to be a language where the users can develop the desired automated behaviors, without having to be concerned about the platform or the devices. The ease of use and the bounding between elements, will be a point where a major effort is justified, since this DSL is intended to be used by non-programmers [9], so the research question is:

Is it possible to define a usable DSL for Home Automation that is able to deal with complex behaviors and devices' conflicts? Additionally, can we schedule those behaviors to be active at a specific time?

1.3 Solution overview

To answer the research question, the Figure 1 was created and represents a possible solution.

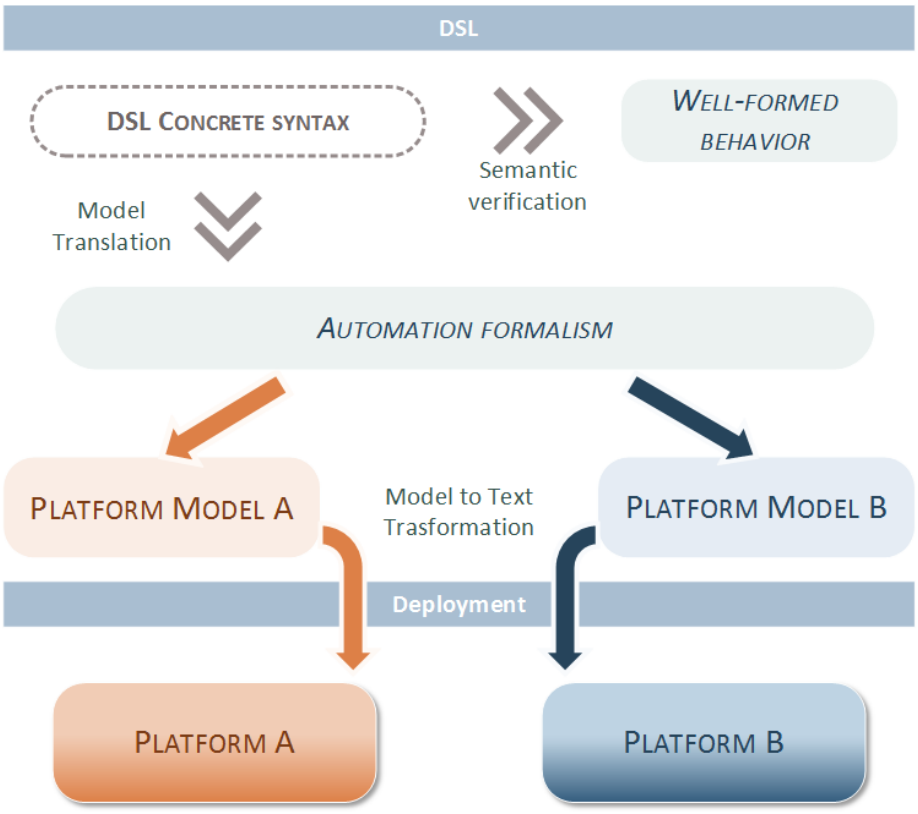


Figure 1 – Overview of the proposed solution

In Figure 1, is possible to see the steps that will be followed towards the implementation of the DSL. In the DSL layer, there are several members of the DSL and some transformations between them. The DSL concrete syntax, should suffer a semantic verification to assess if the produced behavior is well-formed. To be deployed to the specific platforms, there is a model translation to the models of the platforms used. From those models, there is a model to text transformation that will produce text to be deployed into the chosen platforms.

1.4 EXPECTED CONTRIBUTIONS

With this thesis and all the research work involved, we expect to contribute with the following:

- State of the art about the available HA platforms;
- A scenario catalogue to be used as a benchmark;
- A DSL designed for the HA domain that is scalable in complexity, which has been validated by real users.

1.5 ORGANIZATION

The remainder of this thesis is structured as follows:

- ***Model Driven Development (Chapter 2) –*** Has a brief explanation about the more relevant concepts in the domain of HA.
- ***State of the Art (Chapter 3) –*** Represents an overview about the work that has been made in the context of the HA domain.
- ***Technologies and formalisms (Chapter 4) –*** The technologies and formalisms that were explored in order to implement the solution.
- ***DomATIC Languge Framework (Chapter 5) –*** Details all the steps necessary to implement the DomATIC DSL.
- ***Language Validation (Chapter 6) –*** All the steps that were followed to conduct the validation of the language.
- ***Conclusions (Chapter 7) –*** The conclusion about the performed work, which includes the contributions and the future work.
- ***References (Chapter 8) –*** The references used through this document.
- ***Appendix (Chapter 9) –*** The images/tables that, in a way, do not fit in the document flow are located in this chapter.

2

MODEL DRIVEN DEVELOPMENT

Model Driven Development (MDD) is a methodology where models are the main object of the development process [10][11][12]. A model is a simpler representation of reality or, in other words, an abstraction of certain aspects of the real world [13]. The MDD approach is based on having several models on different layers of abstraction that are used to describe the system [14].

2.1 EXPLORING MDD

Each of the models is written in a specific language and they are transformed into another models of the same system. There are some examples of these transformations, like model-to-model (M2M) and model-to-text (M2T) [13]. The first one is used to make the translation between models of the same system, which makes them fully compatible [15]. M2T is used to convert the model into a textual representation of it and also to generate the code for a target platform, which will reproduce the created behavior [16].

The great advantage of this method lies in the fact that different aspects of the system can be represented with different models, which means that the system will have models in distinct levels of abstraction [14]. This not only contributes to a better understanding of the system itself, but also provides a separation of concerns of the system's functionality and its implementation on a specific platform [6][17].

To implement a language based on MDD it is important to understand the different elements that compose its description [17][12]:

1. Abstract syntax – The concepts that inspired the creation of the models;
2. Concrete syntax – The specification of the previous concepts with concrete definition;
3. Static semantics (well-formedness rules) – Rules that establish if the concepts are being used correctly;
4. Dynamic semantics (behavior semantics) – Description about what a model means in terms of real world concepts.

These four points are important guidelines to take into consideration when developing a product based on this methodology.

2.2 MDD BENEFITS FOR AUTOMATION

In the previous section some hints about the benefits and advantages of MDD in the domain of automation were mentioned, which will be explained and summarized in this section.

The main benefit is the abstraction introduced by MDD [2][16]. Since it is a method based on models, each one of them can represent different parts of a system which provides several independent units. This means that a system can be modeled using different layers that represent distinct views of it [18][19]. Another advantage is the increase in productivity, because it is easier to implement separate parts of a problem, instead of taking the problem as a whole.

MDD also has the advantage to have reduced time to market, as it is a solution that can generate code faster. Besides this, models are the basis of MDD, having aspects of the real world represented in a simpler way. This fact makes them understandable by non-programmers, which is translated into an effective cost reduction, since specialists are not needed [13].

Regarding quality, it is proved that MDD leads to increased quality and it is much less error-prone than other solutions [20]. This obviously depends on the engine that does the conversion that should follow a correct-by-construction method, in other words, only correct executions can be defined. This is the aspect that should have more development time associated, to successfully support several types of model constructions without errors [9].

To summarize, the MDD technique has many advantages that can be applied to the area of automation. In this domain, there is hardware at the bottom and software that controls it at the top. As seen with the solutions presented in the Section 3.2, the best way to effectively control the hardware is to add independent layers that provides several abstraction levels, making the system more adaptable.

3

STATE OF THE ART

In this chapter there is an overview of the existing solutions which purpose is to solve problems in the Home Automation domain. The first section considers the most known commercial solutions that are currently being used. The section that follows describes the solutions based on the standard of automation IEC61131. The last section, comprises the most successful solutions based on the MDD methodology that share the characteristic of being domain specific.

3.1 COMMERCIAL SOLUTIONS FOR HOME AUTOMATION

One of the main reasons to do this thesis, is the absence of solutions in the domain of Home Automation (HA) that are platform independent. There are some commercial proposals that only work with specific devices compatible with specific protocols, and there are other platform independent proposals based on Model Driven Development (MDD). In this section, the commercial approaches will be explored to provide some outline about their attributes.

The solutions that are platform specific are commercial DSLs or custom home-made solutions. The most known are KNX/EIB[1], LonWorks[2] and X10[3], but there are also custom solutions that are based on prototyping platforms like Arduino[4] or RaspberryPI[5].

3.1.1 X10

The X10 uses a communication protocol that is mostly used in medium houses and it is easy to install and configure. This protocol uses the Power Line Carrier (PLC) to send data and control the devices, which means that it takes the existing household electrical wiring as a form of communication for the devices. The data is sent at pulses of 120 KHz by 1 millisecond with a certain

[1] http://www.knx.org/
[2] http://www.echelon.com/technology/lonworks/
[3] http://www.eurox10.com/
[4] http://www.arduino.cc/
[5] http://www.raspberrypi.org/

codification, which is composed by the address of the device and a command. The specific frequency signal and the address of the device, are the main information needed to send commands that reaches the devices compatible with this protocol [21].

The great advantage of X10 is also its great limitation. The fact that this protocol relies on PLC to communicate is error prone, because the electrical wiring is not stable. There could be signal attenuation, random signals (noise), electrical oscillation and all types of signal breaks. Additionally, an X10 system is not programmable, just configurable. In other words, it is easier for a user with less knowledge to use it, but if someone wants to further explore some functionalities or change a certain behavior, it cannot be done.

3.1.2 LonWorks

LonWorks is based on a network used to control not only buildings, but also to control small objects [22]. The communication protocol is called LonTalk and it is based in the OSI model (ISO/IEC 7498-1). There are two main components in a LonWorks network:

- *Neurons* – They must be present in each of the devices that are to be used in the LonWorks network, since they contain the entire LonTalk protocol stack. Neurons can be considered like a complete system, because they are comprised of CPU, memory, I/O, communications port, firmware and operating system;
- *Transceivers* – Devices that have a transmitter and a receiver. Their function is to connect the Neurons to the type of media that the user wants, like radio frequency (RF).

To implement a LonWorks network, one needs to acquire at least a device compatible with LonTalk protocol (a device with Neurons) and a Transceiver [23]. These two types of devices are expensive and there is another extra cost with the software needed to configure them. This means that the cost to develop LonWorks applications is higher than most of the commercial systems that exist, in terms of device cost and learning curve, since it is recommended to have some sort of training [22].

3.1.3 KNX/EIB

The European Installation Bus (EIB) is a system that provides the management and the control of electric devices in a building. It uses the electrical bus in the building, so the devices can send commands within that space. This system was later integrated in the Konnex (KNX) association, alongside other previous standards, like European Home Systems Protocol (EHS) and BâtiBUS.

The KNX/EIB is now a standardized protocol (EN 50090 and ISO/IEC 14543) that is fully compatible with the mentioned systems. It is a distributed system that does not require a main controller and all the devices connected to the data bus have their own microprocessor [24]. It supports many communication protocols besides the ones that are used in the electrical bus, like RF and Ethernet, and because of that, this is the most used protocol in Europe.

Despite its great advantages, the KNX/EIB has some problems like the fact that the devices should be compatible with the protocol. The devices also should have their own microprocessor, which raises the prices. Another thing that is not so trivial, is the process of installation, because the user has to choose the form of communication and has to set many parameters until the system is fully functional.

3.1.4 Custom solutions

Some experienced users reject the use of packaged solutions like the ones mentioned in the previous sub-sections, since they want to have more control above the system. Those, are users that use prototyping platforms that consist in a board with a microprocessor that is programmable, or at least, can execute instructions. A tiny device like this has the power to be the core of an HA system, since it can manage communication between the sensors and the actuators. The most known platforms are Arduino and RaspberryPI and their popularity has been increasing, since they are cheaper than a computer and easier to develop on.

These devices do not stand as a viable solution for non-programmers, because of the complexity when there is the need to create a HA system that has the characteristics mentioned in the previous chapters.

3.1.5 Problems with commercial solutions

In this section, we have examined the most common solutions that are available nowadays. The presented solutions provide a way to control some elements in a house, but they do not offer an approach that allows to control the house as an integrated system. This means that the controlled components do not make part of an organized structure, since they are treated as individual parts.

The problem stated above, prevents the creation of more complex behaviors that are not possible to be defined when using the proprietary software that controls the devices. Also, the software is meant to be used by programmers or experts in the specific device, which excludes the end-users.

Table 1 – Commercial solutions' features against the concepts of Home Automation.
(● - Full implementation; ◑ - Partial implementation; ○- Not supported)

Home Automation concerns	*X10*	*LonWorks*	*KNX/EIB*	*Custom solutions*
Platform independent modeling	○	○	○	Depends on the programmer
Graphical notations	○	◑	◑	○
Design reuse	Hard	Medium	Medium	Depends on the programmer
Distinct views of the system	○	◑	◑	○
Integration with other domains	○	○	○	Depends on the programmer
Modularity	○	○	○	Depends on the programmer
Support querying devices	○	◑	●	●
Capacity to extend functionalities	○	○	○	Depends on the programmer

From the Table 1, there are several facts that are important to examine. The first one is the lack of independence between the platform dependent approaches and the devices they control. In the case of X10, there are not any separation between the platform and the devices, since this was the first platform created. LonWorks and KNX/EIB are really similar, but they are also very constrained by their proprietary connection to the devices. The custom solutions are very dependent of the programmer's choices, which could be simpler or more complex. Additionally, as one can observe, those solutions do not follow almost any of the concepts of the HA domain.

A platform independent approach based on a DSL is the adequate solution to a HA system, because it supports the main concerns in this domain. This type of DSL does not require specific devices with proprietary protocols and offers the possibility to easily extend its functionalities, since it can work using modules that it combines. Moreover, it supports graphical interfaces that are appropriate for an intuitive use and reuse of previously defined systems. It can also have different views of the system, which makes it adaptable for most of people, being experts or non-experts.

3.2 THE STANDARD LANGUAGES FOR AUTOMATION

There are some solutions that are inspired in the automation standard IEC 61131 [25] to solve the problems in the automation domain that the devices presented in the Section 3.1 are not capable to surpass. This standard is a general framework for all Programmable Logic Controllers (PLC), so they can incorporate mechanical, electrical and logical aspects [26].

3.2.1 IEC 61131

The framework is composed by a group of five programming languages used to develop software for PLCs. Those languages, represented in Figure 2, are as follows [18][19]:

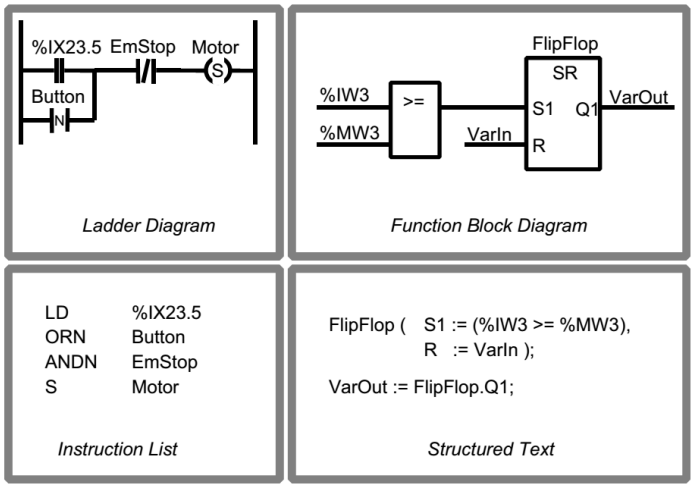


Figure 2 – Languages of the standard IEC 61131-3, taken from [18]

- *Function Block Diagram (FBD)* – It is a graphical language inspired on the domain of the signal processing and describes the functionality between input and output variables. It is represented by blocks and connection lines that are associated to each other, where the blocks are the functions that have the programmable logic. The reading of the data flow is interpreted from left to right and only inputs and outputs compatible can be linked;

- *Ladder Diagram (LD)* – Like FBD, it is another graphical language, but this one was inspired on the field of electromechanical relay systems. It is used to describe how the power flows through the network, since its representation is based on circuit diagrams;
- *Instruction List (IL)* – It is a textual programming language that resembles assembly, since it is low-level and based on instructions. It works as an intermediate language to which the other languages are translated to;
- *Structured Text (ST)* – It is another textual language like IL with a different purpose, because this one is treated as a high-level language. ST does not have machine oriented operators like IL and has a set of compound programming statements, which makes it comparable to PASCAL or C;
- *Sequential Function Chart (SFC)* – It is a language that is both graphical and textual and it is used to divide a complex problem in smaller units. These units are programmed in the other languages of the standard, which means that SFC is used as a manager of communication between languages. SFC is not a programming language *per se*, because it is not possible to create a program with it only, however, it simplifies some aspects of the program. This language is based on a methodology of some well-known paradigms, like Petri-nets and state machines.

3.2.2 An MDD approach to the standard

With these presented languages, there was the need to have a way to associate them together. The point of this association is to avoid inconsistencies between the vendors and also to promote reusability [16][27]. There are two authors that try to achieve this through a MDD approach whose work is worth mentioning.

In [25], the authors try to apply the MDD approach to the automation standard IEC 61131 and IEC 61499, to assess if it is possible to make use of the reusable potential of the MDD approach. To do this, they developed an example that was called the Sorting Machine. The goal is to compare both techniques, using the current design methods and using a new modeling process based on MDD.

The conclusion was that using the current methods makes the generated code extremely hard to reuse. There is no separation between the logic and the implementation, which is reinforced with the lack of a clear place where the software logic ends and the hardware begins. The way they found to work around this problem was to map MDD to the standards, so they could have a concrete separation between the elements involved. This makes the reuse of components more clear and flexible for future applications and makes easier to identify the places where someone can implement specific modules.

The second work is the implementation of an open-source tool similar to an Integrated Development Environment (IDE) for the IEC 61131-3 [26]. This IDE is composed by a Graphical User Interface (GUI) and a backend compiler. Its objective is to reduce the inconsistencies that remain between the implementations of different vendors, because they use the same standards but different file formats. This tool is also important to free the users from the vendors' commercial solutions, since they are very expensive. With this tool, the user has a graphical editor where he can program in each of the previously mentioned five languages, which promotes the consistency when it is necessary to have conversions between them. To sustain even more the claimed consistency, the backend compiler has a lexical analyzer, a syntax parser and a semantic checker that makes the programming less error-prone.

As one could see, the MDD is being used in the automation field, specifically when dealing with the automation standard IEC 61131, because of the several advantages it provides. The abstraction layers have a huge importance when using this approach, since they potentiate the separation of concerns between the software and the hardware. This separation provides more variability in the hardware choices, since it is not necessary to pick only a specific vendor. Alongside this, the top layers do not need to be attached to hardware concepts, allowing constructions much more abstract and user oriented.

3.3 MDD SOLUTIONS FOR HOME AUTOMATION

The MDD solutions mentioned in the Section 3.2 are more oriented towards automation in general, while the commercial solutions shown in the Section 3.1 represent products that are used in the current industry. In the first one, we have the MDD approach to control the devices and in the second we have the companies that produce the devices. With those two information, there is the need to research about solutions that make usage of the MDD methodology to control the previously mentioned automation devices. We have seen that this approach is beneficial in many aspects of this domain, so those benefits would be the same for the sub-domains of automation.

The focus of this thesis is Home Automation (HA), which is a sub-domain of automation. A system of this domain has the control of devices at home and offers the possibility of defining programmed actions when certain events occur [28], meaning that the benefits of MDD previously mentioned can be applied. Generally, HA systems have a graphical interface that offers an intuitive usage to consumers that are not experts in this domain, and could also give the possibility to control the devices remotely [29].

The concept of a HA system goes beyond the simple device control, since nowadays, the main concerns are the comfort, the security and the energy savings [30]. A system should provide to its user the possibility to reduce the time spent doing a task; a way to monitor and prevent misfortunes within the house and also minimize the equipment energy consumption [31].

The difference between automation in general and HA, is that in HA we are addressing a particular domain that has elements specifically used there. To manage those elements, it is needed an additional abstraction layer that consists in a Domain Specific Language (DSL) to handle problems restricted to HA.

3.3.1 Domain Specific Languages

A Domain Specific Language (DSL) is a type of programming modeling language that is designed specifically for a domain. Opposing to a General-purpose Language (GPL) that is intended to be applicable across domains, a DSL is focused to express terms in the domain of the problem instead of terms of the computational solution. DSLs have emphasis on usability concerns, represent domain concepts and are used to raise the abstraction level [6][7].

To develop a DSL, several steps are detailed in [9]. The first one is the Domain Analysis, since the DSL has to take into account the particularities of the domain that will be explored, like the terms and expressions intrinsic to it. When all the main terms are gathered, a metamodel should be developed, which makes it the second step. The third step is the actual implementation of the DSL, based on the metamodel previously defined. The final step is the validation of the DSL in the context of the domain with users. This final stage has great importance since validating the language with real users, will be a test to the work done. From this evaluation, there will be

conclusions to make and they will be the starting point of the DSL next step, which is review all the steps knowing the results of the experiment [32][33].

From the above points, we can see that this process is an iterative one, since to be flawless, a DSL must be redesigned several times (Figure 3).

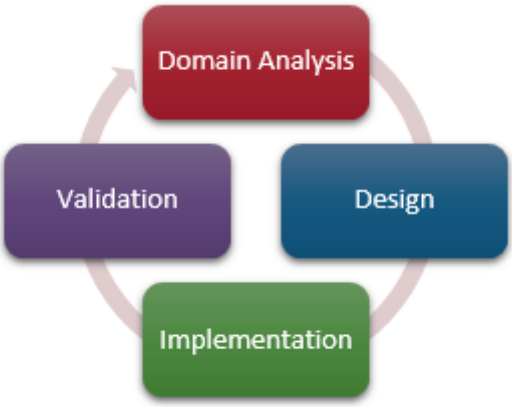


Figure 3 – The DSL implementation cycle

DSLs are becoming a concept with great importance in software engineering, since nowadays people want specific solutions for their needs, aiming to increase their productivity. With the increasing demanding for DSLs, the tools to develop them are improving as well, which means that is even easier to build a DSL.

In the particular case of HA, a DSL can be used to ease the implementation of a solution for this domain, as the Section 3.3.2 shows.

3.3.2 Languages for HA

There are some solutions for the HA domain, where the authors suggest the combination of the MDD methodology with DSLs.

For a general implementation of a DSL for HA, the work presented in [34] was considered, which consists in a case-study based on a smart home. In this example, Vöelter is focused in the two first steps of the development of a DSL (defined in Section 3.3.1), namely, the domain analysis and the definition of the metamodel. This work is purely anecdotal, because there is not an actually implementation of the DSL, however it is still interesting to study, since it presents in detail two of the first main steps.

In a more practical context of DSLs in HA, the most notorious works came from Clemente, P., et al. [2] (Figure 4a), Prähofer, H., et al [35](Figure 4b) and Sanchéz, et al. [1](Figure 4c). The first authors have developed a software architecture that bounds together the concepts of DSL and HA; the second ones made a DSL for programming event-based automation solutions; the later wrote a doctorate dissertation where he describes his system for HA in detail.

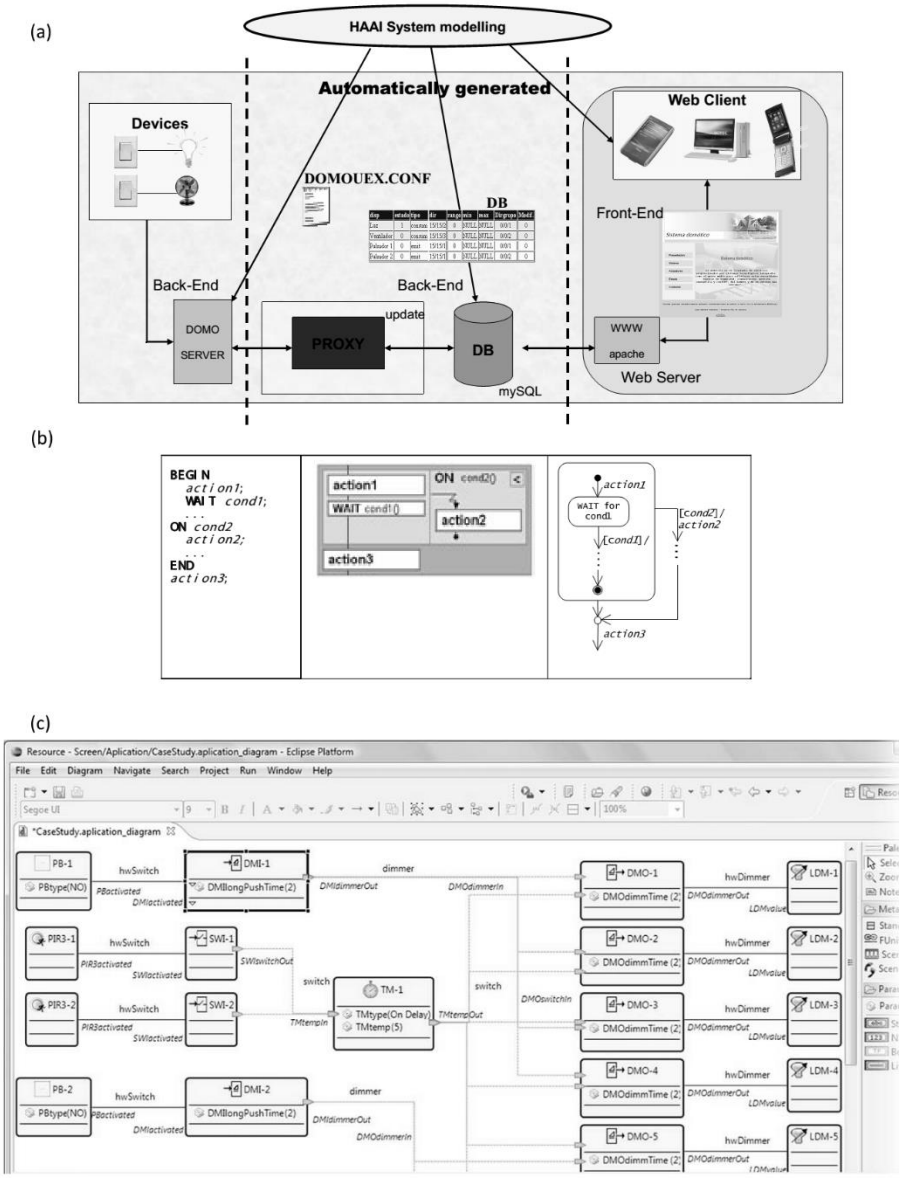


Figure 4 – DSLs for Home Automation, taken from [2]*(a),* [35]*(b),* [1]*(c)*

In the publication of Clemente, P., et al. [2], he follows practically the same steps as Vöelter. Their DSL is focused in HA and Ambient Intelligence (AmI) systems, and their approach also has a metamodel as basis. From the metamodel they make model transformations to create a systems' back-end and front-end. The system is graphically represented by the DSL and it was developed to support the addition of more devices. The interest of this work lies in the usage of MDD methodology, the development of a DSL for HA purposes and also in the model transformations. The implementation is not fully functional and just tackles "the complexity of these kind of systems, facilitating the specific domain elements and the rules required to model these systems" [2].

While the previous work is focused in testing what is the impact of the MDD methodology with DSLs, the next one presents the Monaco language [35]. The main purpose of it, is to develop a way to simplify the programming of automation devices, so it can be accessible to domain experts and end users.

Lastly, HAbitATION describes a DSL for HA using the MDD technique [1][36]. The approach is the same used by the previously mentioned authors, but what is interesting is that he goes further into

the implementation of a DSL for HA. While the focus of the Monaco language is to simplify the programming of automation devices, the goal of HAbitATION DSL is to provide a way for the domain experts to control the devices without programming them.

The opinion issued by each one of these authors is unanimous about DSLs. They help to raise the abstraction level to a point that it makes easier to manage a system and also for non-specialized people to build their own. Graphical DSLs are recommended, since they have a substantial gain in expressiveness and ease of use, even for people that do not have many experience in the use of personal computers for technical purposes. Other benefits of DSLs include the reduced time-to-market and their maintainability [37], which is a valuable point in the context of HA.

However, there are some problems when one is implementing a DSL for HA. The analysis of this domain is hard to perform and there is not a wide choice of tools to develop a DSL. To be successful, a DSL for HA has to be very clear about the aspects of the domain and how the results of the domain analysis can be used in the DSL design and implementation.

Both the Monaco and HAbitATION languages provide the most interesting work around the combination of MDD and DSLs. For this reason, it is important to analyze in more detail each of them, to study the techniques used and to identify the aspects that can be improved.

3.3.2.1 Monaco DSL

As mentioned before, this language aims to provide an easier way to program automation devices. To do this, the authors made an effort to keep the language simple, so it could be used by domain experts and end users [35].

This language is similar to Statecharts (Section 4.1.1) in terms of expressiveness, but it adopts an imperative notation, like procedural abstraction, synchronous procedure calls and high-level language syntax. The other features offered by this language are its asynchronous event handling mechanism, the hierarchical component and communication architecture and the static nature of its programs.

Although they claim it to be adequate for end users, the authors did not make any type of test that supports this statement. The language has a strong influence of the Statechart formalism, which contains programming concepts that are not appropriate to be used by non-technical people [38]. Despite this, the developed approach can be interesting to be used by domain experts at a level where they need to control some aspects of the hardware in a more abstract way.

3.3.2.2 HAbitATION DSL

The HAbitATION language is a well-developed example of the usage of DSLs in the HA domain. In this work, Buendía has followed all the steps required to make a DSL, beginning with an extensive analysis of this domain.

First of all, it is important to explain how he structured his solution. For Manuel Buendía, the devices are called Functional Units and they have parameters which define their behavior. To connect a Functional Unit to another, there are Services that can be the same for different Functional Units [36].

This fact leads to the creation of a Functional Units' catalogue that has their definition and Services. This catalogue supports the addition of more devices, distributed by categories, and

has two main groups: Passive Units and Controllers. Passive units are Sensors and Actuators. Controllers are elements that require some programming logic to implement a certain behavior.

Another relevant feature is the creation of Scenes, which are sequence of events that are triggered in order. Scenes are meant to aggregate a sequence of events allowing the user to reuse them as needed.

The solution of this author has also the support for different views of the system. He distinguishes between a view where the user can define the parameters of the Functional Units and a view where it is possible to identify the distribution of the Functional Units through the house.

The HAbitATION DSL represents a significant step towards the control of automation devices. However, the author identified some problems with his approach [1]. The first one is the overlapping effect that can occur when two opposite behaviors coexist. He solves this problem by classifying the devices by requirements, which is a vital task that is done manually by language experts and, therefore, error-prone. The other problem is the lack of a method to test the defined behavior without deploying it to the target platform. The author admits that a simulation platform/animator is something that could be extremely useful to validate the models beforehand.

3.3.2.3 Potential for improvement

Taking into account the two last languages detailed in the Section 3.3.2, one could see that they represent two distinct ways of controlling automation devices. The Monaco language is more oriented towards programmers and domain experts that are proficient in programming. Alternatively, HAbitATION is meant to be used also by domain experts, but they do not need to know about programming to control the devices. In both of the DSLs there is room for improvement, which does not mean that they are not valid languages. In fact, if the two of them are combined and some particular flaws are corrected, we eventually end up with a solution that is adapted to technical people, domain experts and possibly end users all together.

We have seen that the HAbitATION had the problem of not having a simulation platform to validate the models. On the other hand, the Monaco language provides a Statechart-like approach that proved to be appropriate to illustrate how devices work. Additionally, HabitATION is more oriented for domain experts only, due to the overlapping effect (Section 3.3.2.2), which is only avoided manually by the users of the DSL, who have to be specialists in the domain.

The Table 2 summarizes the comparison between the two platforms.

Table 2 – Comparison between Monaco and HAbitATION DSLs
(● - Full implementation; ◑ - Partial implementation; ○- Not supported)

Concerns	*Monaco*	*HAbitATION*
HA concepts	●	●
Model reuse	◑	●
Graphical notations	◑	●
End-user oriented	○	●
Well-formedness rules validation	○	○
Behavior simulation	○	○
Behavior deployment	●	●

Considering that HAbitATION is a DSL addressed to domain experts and Monaco is a DSL more oriented to programmers, there is the need to have an appropriate solution to common home users. Regarding the best practices used by the other two DSLs, a DSL for non-technical people should have three main components: validation of a defined behavior; simulation of that behavior and the generated code for the target platform. To define a behavior, a graphical DSL can be constructed with general concepts of the HA domain, which can then be transformed into simulation or deployment. The simulation is appropriate to be used by domain experts, so it can be defined using the Statecharts paradigm, whereas the deployment is represented by code to be analyzed by programmers. The end user can avoid both the simulation and deployment phases, simply by defining his case, verifying it through well-formedness rules and send it to the target platform.

4

TECHNOLOGIES AND FORMALISMS

In this chapter there is a discussion of the tools that can support the implementation of the DSL. There is the need to choose an adequate automation formalism that can support HA concepts. On the other hand, in order to implement the actual language, a set of tools are required to provide a GUI and the logic behind it; a simulation platform to evaluate the created behavior; an execution platform to test the behaviors in real life.

4.1 AUTOMATION FORMALISM

The DSL that will be developed, requires a strong support model that should specify all the features that needs to be expressed, such as behaviors, conflicts, decisions and other concerns. Therefore, the chosen visual modeling formalism should provide a way to describe all the features needed in the HA domain.

The following scenario will be used throughout the following subsections, as a common denominator for the technologies:

Heating room Scenario

Sensor: Temperature sensor

Actuator: Heater

Description

The temperature in the room can change in a natural way, influenced by the temperature outside or even the presence of people. With this in mind, it can be said that the room can be in one of three states: Cold, Comfortable or Hot. The Comfortable state corresponds to a state where the room is considered to be between two temperatures (e.g. 20°C and 25°C), where the users do not have the necessity to change nothing. When the room is in Cold state, it means that the system should turn the heater on, so the room can evolve to the Comfortable state. If the room reaches the Hot state, the heater should be turned off immediately.

There is, however, a characteristic in this particular heater. It can only work for a while (e.g. a straight hour), and then it enters in a pause mode (e.g. thirty minutes) to cool down, even if the room has not reached the Comfortable state.

4.1.1 Unified Modeling Language (UML) – State Diagrams

UML is a general-purpose language that is widely used for the design of software systems [39]. This language is not domain specific, since it provides a syntax that is used to model all kinds of systems. To offer more specific tools, UML has profiles that are designated for a particular domain, like class structure, state based behavior or deployment [9]. However, these profiles are still general-purpose, since they just address a specific software domain and not an element specific domain.

To serve as basis for implementation of the State diagram, a Class diagram was created (Appendix, Section 9.2, Figure 46)

State diagrams (SD) are "a set of concepts that can be used for modeling discrete behavior and reactive systems through finite state-transition models" [39]. The idea behind SD is that there is a controller that can be in different states. When this controller is in a particular state, there could be a set of triggered events that forces a transition to another state. This means that a sequence of events is used to progress from state to state [40].

Using a profile oriented for Modeling and Analysis of Real-Time and Embedded Systems (MARTE), it is possible to model a HA system using SD. The MARTE profile offers support for the development of Real-Time and Embedded Systems (RTES), which means that the notation used by MARTE can be used to model a SD for each device.

Figure 5 exemplifies the usage of a SD model in HA using the MARTE profile, based on the scenario defined in Section 4.1.

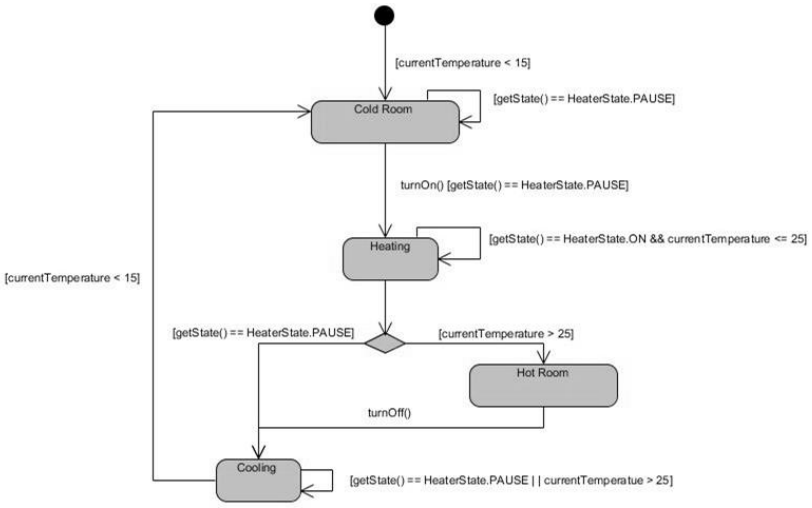


Figure 5 – State diagram of the Heating room scenario

4.1.2 Petri nets

Petri nets are a graphical and mathematical modelling language used for many systems [41]. A Petri net is a directed bipartite graph composed by transitions and places. The first ones represent events that may occur and the others represent conditions that guard the passage to the next node [42]. There are also arcs that run from a place to a transition or vice versa, but there cannot be

defined arcs that run between places or transitions [39]. Another element of the Petri nets are the tokens that are used to simulate the dynamic and concurrent activities of systems.

An example of the usage of a Petri net is present in Figure 6.

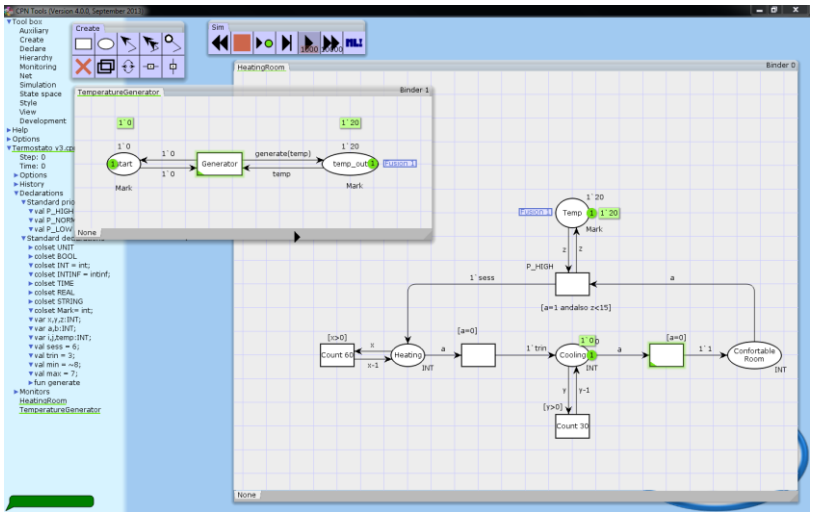


Figure 6 – Implementation of the Heating room scenario using Petri nets

4.1.3 Usability comparison between State diagrams and Petri nets

Petri nets can be applied to a variety of applications, but there are some problems when one needs to describe a system/scenario. Petri nets offers several ways to model a system and because of that, there are many ways to interpret the result [41]. Another problem is the complexity of the models, as Petri nets become too large, even when representing a simple model [41]. When implementing the Heating room scenario, the perception is that Petri nets were too complex to be used in the domain of HA, since they can grow in unpredictable ways. This fact will difficult the entire process of developing a DSL, since it is desirable to ease the modelation process and keep it simple.

On the other hand, the usage of SDs was more intuitive, because the syntax provided is more than enough to represent the desired scenario and can be used to model the entire system. There is another advantage of using the UML notation, which is the compatibility with the tools that will be used to develop the DSL. This point will be more evident in Section 4.2, since they use a very similar notation that makes the transition from SD notation to the tools' notation easier.

4.2 Tools to develop the DSL

There are several tools that support the development of a DSL, but there are some reasons to exclude them, which are detailed in the following list:

- *MetaEdit+*[6]

This is a tool that supports many functionalities, such as multiple users, multiple projects and runs in several platforms. The main problem is that this tool is used for commercial purposes, therefore, there is the need to acquire a license. This limitation is enough to exclude this tool, since we do not want to use paid alternatives, unless it is absolutely necessary.

[6] http://www.metacase.com/mep/

- *GME*[7]

GME stands for Generic Modeling Environment and is a modeling toolkit oriented for domain specific environments. The concerns about this tool is its platform dependency (just works on Microsoft Windows) and its lack of community support.

- *AtomPM*[8]

This framework is a little bit different, since it permits the generation of DSLs that run in the cloud. Those DSLs are also made through a web interface, making it independent from the operating system. There are however some uncertainties about this tool, because it is very recent and has a reduced support community.

- *Epsilon*

Epsilon proved to be the most complete platform, since it has several years of development and improvements, has a great community and is open source. Epsilon is a DSL development tool developed by the Eclipse Foundation9 and is a family of languages and tools that are oriented for code generation, M2M transformation, model validation and others (Figure 7).

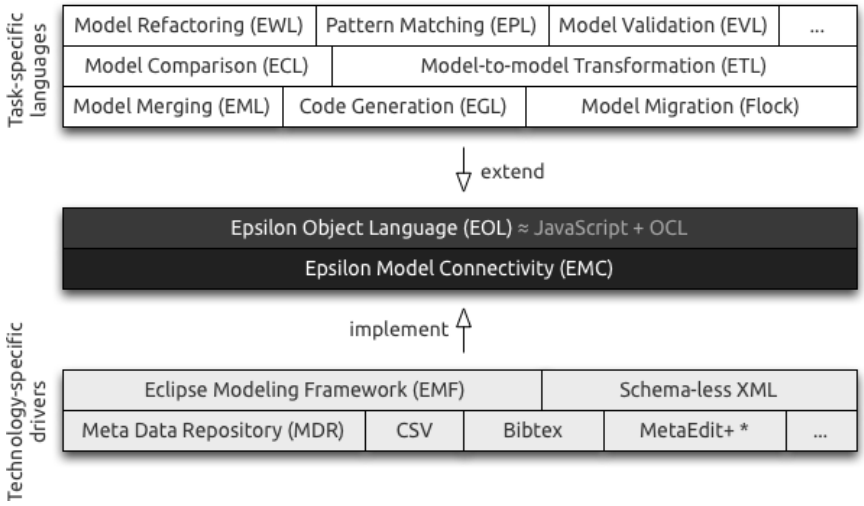


Figure 7 – Epsilon's tools and languages overview

Since this platform is composed by several tools and languages, the following list has a selection of which ones are adequate to the implementation of the aimed DSL:

- *The Ecore Model*

  It is a model that represents models in EMF. Ecore is an XML-scheme to define object models, which makes it one of the main concepts in Epsilon. All the languages and tools should work with Ecore, since it is the model that provides the compatibility between them.

- *Eclipse Modeling Framework (EMF)*

  EMF is a modeling framework that is used to build tools and applications, based on a structured data model, which is defined in Ecore. EMF is another core feature of Epsilon, since it provides the interoperability with other tools and applications based on EMF.

[7] http://www.isis.vanderbilt.edu/Projects/gme/
[8] http://syriani.cs.ua.edu/atompm/atompm.htm
[9] http://www.eclipse.org/epsilon/

- *Graphical Modeling Framework (GMF)*

  GMF is the same as EMF, but for graphical purposes and it even uses the EMF as a component for the definition of the application. To generate the rest of the necessary models for GMF, Epsilon provides EuGENia. This tool generates in an automatic way three types of models used by GMF: .gmfgraph, .gmftool and .gmfmap.

- *Epsilon Object Language (EOL)*

  EOL is an imperative language that is used to create, query and modify EMF models. Because it is an imperative language, EOL has some of the OCL's features, as well as some of the imperative singularities of Javascript.

- *Epsilon Transformation Language (ETL)*

  ETL is a M2M transformation language, which offers the standard features provided by a general transformation language and even has the possibility to navigate through the models. This language provides the definition of rules and the execution of schemes and because it is based on EOL, can make use of imperative rules in complex models.

- *Epsilon Generation Language (EGL)*

  EGL is a M2T language that generates code and other textual items from models. Like ETL, this language is built over the EOL, which means that it makes a reuse of some features, like making model inspection or controlling the program flow.

- *Epsilon Validation Language (EVL)*

  EVL is also based on EOL and it is used to perform the validation of metamodels about their consistency and to make some repairs in case of necessity. This language can be combined with GMF/EMF, which means that the validation is done in the context of their editors and that the errors are generated as they should be.

With all the tools and languages detailed, it is possible to know how the DSL will be implemented. The GMF editor provides a front-end based on an Ecore model, modeled using the concepts of HA. The ETL language will be used to make a M2M transformation that could be useful if there are other tools involved in the process. The EGL language generates code that will run on the devices, based on the Ecore model. To assess if the models defined by the user are correct, the EVL is used, since it has a set of well-formedness rules to alert for possible errors.

4.3 MODELING PLATFORMS FOR SIMULATION

A DSL built for the purpose of HA should have a simulation platform as support for the verification of behaviors. That simulation platform can be a tool that is usually known for the purpose of modeling and simulation. If a solution is found with the usage of a tool that is acknowledged in the field of simulation, it is a better choice than develop the same functionality with general-purpose modeling languages [43].

The chosen tools were LabVIEW (Section 4.3.1) and Simulink (Section 4.3.2), since they are widely known and have the required characteristics to be used in this domain. To assess whether these tools are appropriate, there will be used the Heating room scenario defined in Section 4.1.

This scenario has some features that makes it a good choice to implement using the two modeling mentioned tools, since the heater has a very particular mode of operation that has an impact in the whole simulation. The following values and specifications will be used in the implementation:

Heater implementation

The heater can be in one of three states (OFF, ON, PAUSE), so a state for each one of them was created.

The transitions from one state to another were implemented as follows:

- OFF -> ON – To evolve from OFF to ON, the heater should be turned on explicitly. This fact means that some element should do this action. To accomplish it, it was created a shared Boolean variable (HeaterON), which is used by other elements that desires to turn the heater on.
- ON -> PAUSE / ON -> OFF – The heater can only be turned on for an hour, so the transition has a guard that counts the elapsed time against one hour. The transition can occur when the time has passed or if the room evolves to the Hot state, where it should be turned off immediately.
- PAUSE -> OFF – The heater must be in pause by half an hour, so the transition's guard is only the verification of the time elapsed.

To simulate the evolution of the temperature, it was developed a temperature simulator that must know in which state the heater is, so it could raise or decrease temperature. It was created another shared variable (HeaterState), that is used to know if the temperature should raise (HeaterState = ON), or if it has to decrease (HeaterState = OFF/PAUSE).

Temperature sensor implementation

The temperature simulator was created within the temperature sensor to make the scenario implementation easier to understand. The temperature sensor has a shared variable (CurrentTemperature) that provides the information about the temperature detected at the moment. This variable will be important to define in which state the room has to be and what actions should be taken to make the room comfortable.

Room controller implementation

In the scenario description, it is considered that the room can only be in one of three states:

- Cold – The temperature is below a minimum threshold.
- Comfortable – The temperature is between a minimum and a maximum thresholds.
- Hot – The temperature is above a maximum threshold.

The objective of this scenario is to have the room in the best possible condition and for the most possible time, that is, in the Comfortable state. There are, however, limitations about the heater, because it can only be on for a certain period of time and has to wait another period before it can be turned on again. To manage all these aspects, the room controller element was created.

This controller has three main states and a dummy state (Check room state) that will be used only to define in which state the room is, depending on the current temperature. The other states have a behavior as follows:

- Cold – It gives the order to turn the heater on, since the goal is to evolve from the Cold state to the Comfortable state.
- Comfortable – This state does nothing, since it is the better state to be. If the heater had more modes of operation, this state could set the heater to operate in a lower temperature mode, for example.
- Hot – If the room is in this state, it means that the heater is warming up the room too fast. When this happens, this state gives the order to turn the heater off, by setting the "HeaterON" to false.

There was a previous statement about choosing tools that should be free to use, but in the case of the simulation there are no viable alternatives to the presented platforms below.

4.3.1 LabVIEW

LabVIEW[10] is a platform that uses a graphical programming language for development and system-design. LabVIEW is a short for Laboratory Virtual Instrument Engineering Workbench and it was developed by National Instruments. This is a tool that produces code that can run in multiple platforms, like desktop systems, mobile systems and even in microprocessors [44], which makes it a platform widely used.

For the purpose of this DSL, it will be used a specific module of LabVIEW, the Statechart Module, given that the chosen formalism was the SDs (Section 4.1). This module provides a modelation paradigm that is based on states, transitions, events and triggers, which are the required features to simulate a HA system.

As stated above, the Heating room scenario was modeled using LabVIEW. To implement this scenario, one should pay attention to several details, such as the identification of elements involved, their behavior and the dependencies between them. In this particular scenario, there are three main elements: the temperature sensor (Figure 8), the heater (Figure 9 and Figure 10) and the room controller (Figure 11), which is responsible for the interaction between the other two. Since there are three elements, there will be three state charts, one for each element.

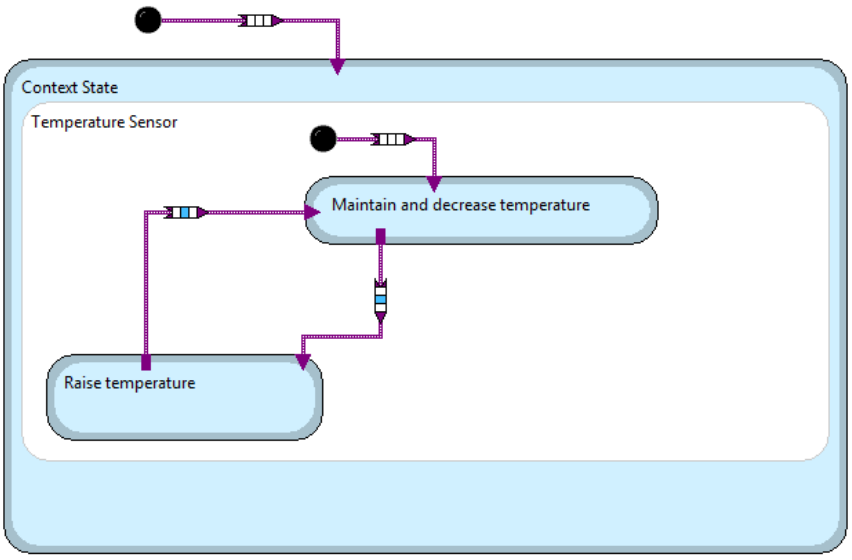


Figure 8 – Statechart that simulates the temperature sensor

[10] http://www.ni.com/labview/

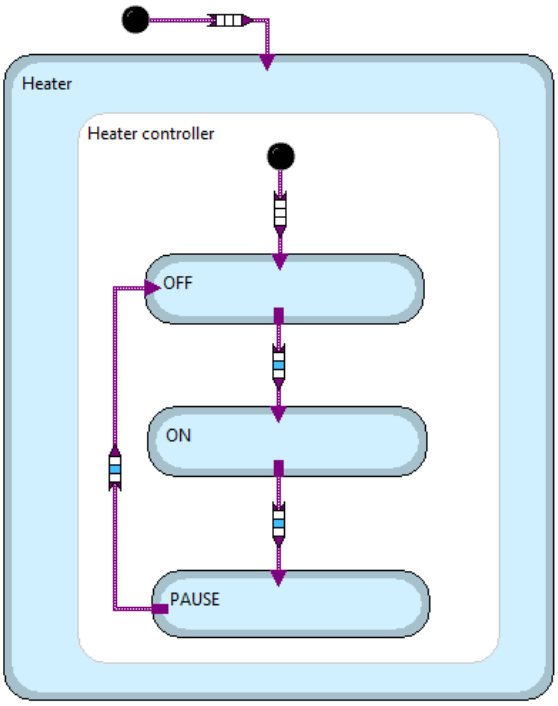


Figure 9 – Statechart representing the heater

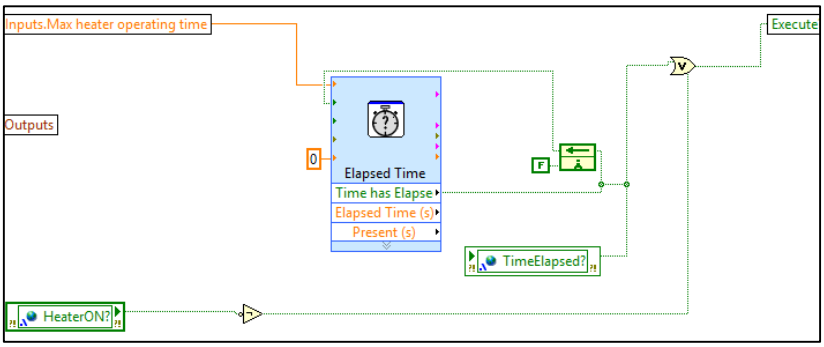


Figure 10 – Implementation of the transition's guard ON->PAUSE condition

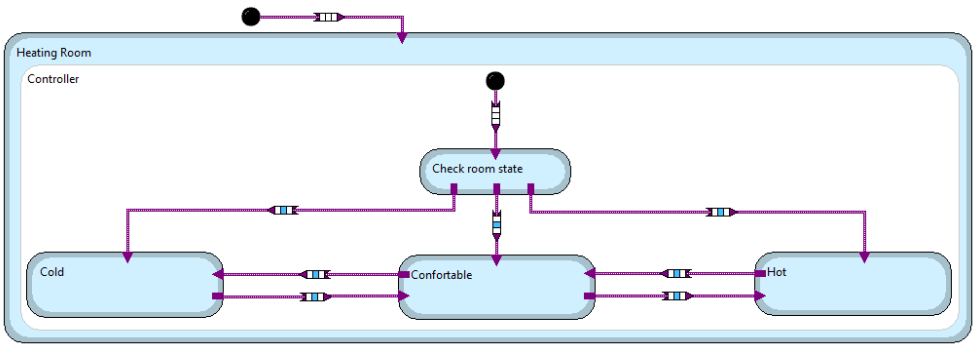


Figure 11 – Statechart representing the room controller

The Statecharts have some variables in common, like the temperature, which should be created as shared variables [44]. This aspect has great significance, because this is one of the details that make possible the usage of LabVIEW in the context of HA.

The dashboard of this scenario in LabVIEW can be found in Section 9.3, Figure 47.

4.3.2 Simulink

Simulink[11] is a block diagram environment for multidomain simulation and Model-Based Design. It is a platform developed by MathWorks and supports modeling, simulation and analysis of dynamic systems. It has a graphical interface and supports the addition of libraries. Since this tool is embedded in MATLAB, it can either produce code for MATLAB or be scripted directly from it. It has great support for a variety of hardware devices and produces code in C or C++.

Much like in LabVIEW, it was used a specific module to represent the scenario, which was the Stateflow library. This library has support for using Charts that can represent the elements identified for the scenario (Figure 12, Figure 13 and Figure 14).

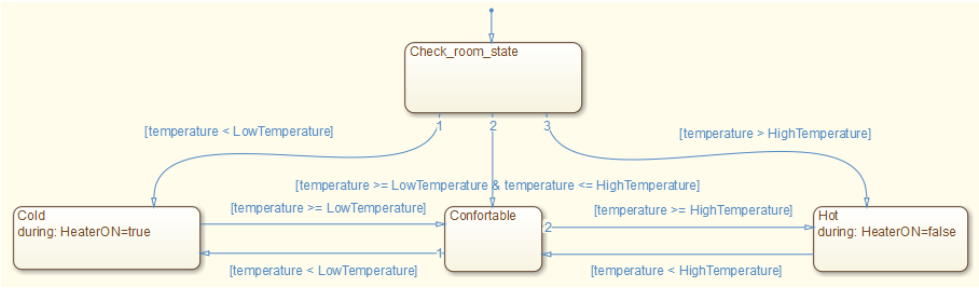


Figure 12 – Stateflow of the room's states

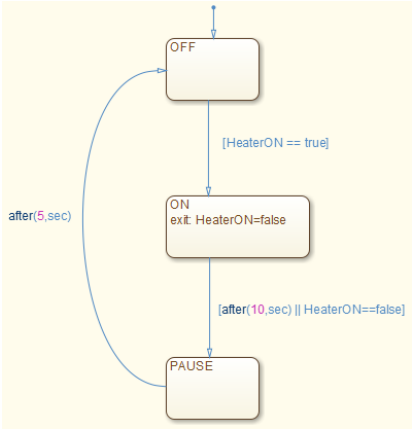


Figure 13 – Stateflow of the heater

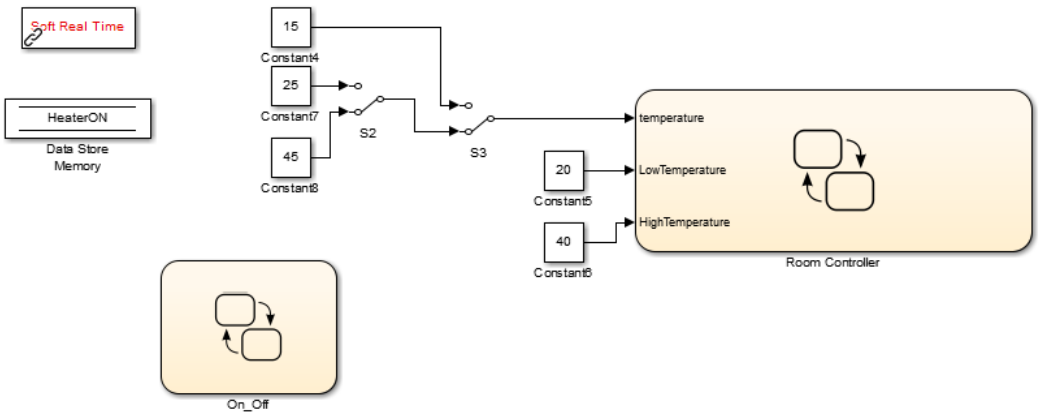


Figure 14 – General implementation on Simulink

[11] http://www.mathworks.com/products/simulink/

This two tools are identical, so the same logic used to model the scenario in LabVIEW was used for Simulink. Instead of having shared variables, Simulink calls them Data Store Memory. To represent the elements, a Chart were created for each one, maintaining all the features described in the beginning of this section.

To make the simulation run in real time, an external library (Soft Real Time) was used, because when modeling in Simulink it must be chosen a platform as the final device. This fact makes the match between the real time and the simulation time more accurate, since there are several parameters to control the simulation in terms of speed.

4.3.3 Comparison between LabVIEW and Simulink

The scenario was implemented with success using both tools, but there are some details that make a difference when one is modeling using LabVIEW or Simulink. Before explaining those differences, it is important to analyze Table 3 to be aware of the features that these two platforms offer.

Table 3 – Features supported by LabVIEW and Simulink

Features supported	*LabVIEW (Statechart Module)*	*Simulink (Stateflow Library)*
Language expressiveness	The models can be defined in a graphical way, so is the logic behind them. Really easy to use.	The major models are graphically defined. The logic must be written with text, as a standard programming language, which provides more control.
Ease of modulation	All elements are modules that can be linked with each other if they are compatible.	All elements are modules that can be linked with each other if they are compatible.
Simulation options	Offers visual appealing simulation options, with a front panel that looks like a final GUI. The simulation is also easy to control.	It has the possibility to simulate the models and control the simulation time.
Code generation	It can generate code in C.	It can generate code in C and C++.
Modules/library add-on support	Supports the addition of custom made modules.	Supports the addition of custom made libraries.
Script support from external tools	Does not support the loading of scripts or other types of modules from external sources.	Supports the generation of a whole simulation case through external scripts.
Documentation	Has some documentation from NI and good support from the community.	The documentation is almost only from the Mathworks. Poor support from other sources.

Both tools have almost the same functionalities and both of them are suited for the domain of HA. There is however a major difference that makes Simulink more adequate than LabVIEW.

The first one is the simulation time. Simulink has the problem that the target device must be chosen in the creation of the project, but this fact was revolved using an external library that proved to be extremely useful for controlling the simulation.

Another feature that distinguishes these two tools is the views provided by the tool. Both have a block diagram view, where the simulation is executed, but LabVIEW adds another view which is called “control view”. This view is like a GUI that offers the possibility to control the simulation by setting the input parameters and by modifying the variables in real time.

LabVIEW has the advantage in the documentation material. Mathworks website is very complete and have many examples about how to use their tool, but there is a lack of material in external sources. NI website offers a little less support, but that is complemented with the help and examples from the community. In the other hand, Simulink can load script files that can be generated from external sources, while LabVIEW cannot. Since we are implementing a language and we are using an external tool to run the simulations, the generated code for the simulation must be loaded to the tool. This is the characteristic that makes the Simulink more adequate to our needs.

The two products are very similar, but the fact that LabVIEW does not support the loading of files from an external source, excludes this tool and so, the choice is the product from Mathworks.

4.4 EXECUTION PLATFORM

The usual expectation by an end user after a behavior defined, is to witness its result replicated in the real world. Since there are several automated devices from the different vendors available, the DSL must produce code that is compatible with them. Considering these characteristics, the end user can acquire any automation device, define a scenario in the DSL and export it without needing to have the knowledge about the communication between the software and the hardware.

As a proof of concept, we considered the usage of a framework that allows the communication with devices from the most known vendors of this domain, such as KNX, LonWorks, DMX and Modbus. The execution platform came from a Portuguese company called Domatica[12], which is specialist in providing equipment for several areas of Automation, including HA. We took advantage of having the material available at the *campus* to do the deployment aspect of this M.Sc. thesis, since the equipment for this domain is very expensive. It would be particularly difficult to accomplish the results shown in the remainder of this document, like the usability tests, without the proper material.

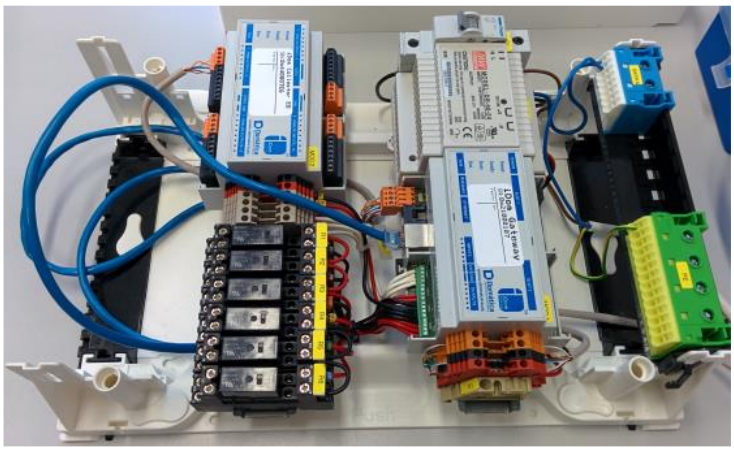


Figure 15 – Domatica's implementation platform

With this device, we can monitor and control the connected devices, based on a scenario developed by the user. This execution phase is adapted to end users, but also to programmers who can analyze and change the produced code in order to adjust it for their needs.

[12] http://www.domaticasolutions.com/

In Figure 15, we can observe the internal appearance of the device. It uses communication gateways that provide a way to monitor the connected devices, which are used for the communication between them. The internal memory is used to store the developed behaviors and keeps them, even after the device has been turned off. The communicatin between the platform's gateways and modules is done using a framework called iDom Framework.

5

DOMATIC LANGUAGE FRAMEWORK

As seen in the previous sections, there is no ideal solution that is suitable to all types of users, so, our purpose is to create a DSL that suppresses the flaws of the analyzed solutions. Additionally, this DSL aims to be used by end users, which neither Monaco (Section 3.3.2.1) nor HAbitATION (Section 3.3.2.2) contemplate. The DomATIC is also prepared to be used by domain experts and programmers as well, since it supports behavior simulation and code deployment. Since we are in the domain of Automation and this theme have a huge quantity of different elements, the focus for this language will be the Home Automation (HA) domain, as can be verified in Figure 16.

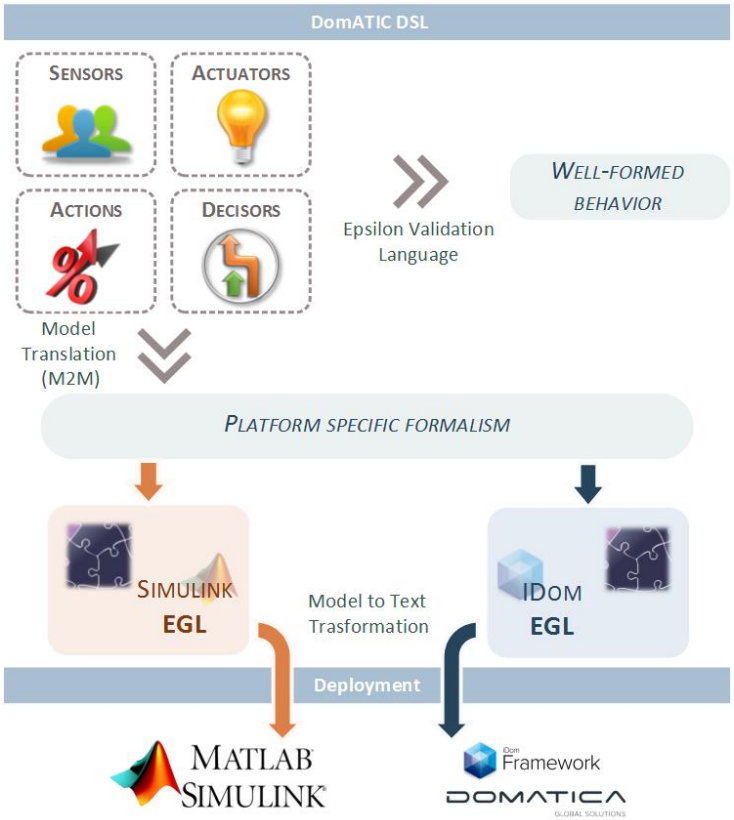


Figure 16 – The DomATIC DSL detailed overview

This subset (HA only) contains a shorter list of typical devices, making the use cases more familiar and perceptible. The implemented DSL has its concrete syntax based on four elements: Sensors, Actuators, Action and Decisors. Syntax is supported by an EVL file that has a set of well-formedness rules that will guarantee that the behavior is in conformance with the rules of the language. From the syntax used, there is a model transformation to a specific formalism that depends on the target platform.

The concrete syntax is based on an Ecore model, which defines how the elements of the DSL are connected and what are their properties and types. From these models, there is a translation to two EGL files that will produce code for two different purposes: simulation and execution. The Ecore model is the basis of the EGL models that were defined for each platform. Both of the EGL models were produced simultaneously in order to avoid differences between the simulation and the execution, since the simulation model will produce code for Simulink (Section 4.3.2) and deployment will create code for the iDom Framework by Domatica (Section 4.4).

This DSL is named DomATIC, which stands for *Domus* Automation Tool for Intuitive Configuration. We thought this name was adequate, since both the name and the meaning have some relation with this domain (*Domus* is the latin word for house).

The remainder of this chapter will follow the DSL implementation cycle defined in Section 3.3.1, where the development process of a DSL as being an iterative procedure composed by four phases: Domain Analysis, Design, Implementation and Validation. The first phase consists in analyzing the target domain and identify the most common terms and concepts, as well as determining possible usage scenarios. The Design phase is where the construction of the DSL structure takes place, since it is here that its representation models are developed. The next phase is the Implementation phase, where the language and the editor are defined, but also the validation rules and the code generation based on the model. The last step is the Validation, which should use the rules defined in the previous phase to validate the construction of the automation case. It also must be tested by real users to identify possible errors in the language and to verify if it is usable and understandable. If necessary, all these phases can be revisited to eliminate the identified faults.

5.1 DOMAIN ANALYSIS

When performing a Domain Analysis, it is necessary to have into account the domain's terms and concepts, which are the origin of the DSL. From there, in order to build several scenarios, some other elements will be added to compose an appropriate language for the target users. As mentioned before, the two primary elements are the Sensors and the Actuators, but there are other necessary elements for the creation of scenarios that must be added, as shown in the rest of this section.

5.1.1 Understanding Home Automation

In order to understand the elements that compose this domain, it was necessary to meet domain experts in HA. Some meetings with students and professors from *Faculdade de Ciências e Tecnologia*[13] (FCT) and *Instituto Superior Técnico*[14] (IST) were conducted, since these two institutes have people that are currently working with the devices mentioned in Section 3.1.

From those meetings, we extracted the typical situations intrinsic to this environment, which will be treated from now on as scenarios. The conception of usage scenarios is vital to fully understand

[13] http://www.fct.unl.pt/
[14] http://tecnico.ulisboa.pt/

the importance of using home automation devices and how they will be used. These scenarios represent case studies that will be used to characterize the typical usage of HA equipment, express the connection between devices and determine their behavior in a practical way.

To keep the emphasis in the DSL, the scenarios and conclusions taken from the meeting will be explained in the Appendix, Section 9.1. This section is intended to provide a brief explanation about the concerns to be taken into account when performing a domain analysis in HA. Some of the thoughts presented do not necessarily mean that that approach should be followed, however, they were an inspiration to develop the DSL.

5.1.2 Sensors and Actuators

Sensors are devices that monitor the environment, like the light intensity, the temperature variation and the movement. The aspects that Sensors monitor are treated like inputs that should be interpreted by a controller. Then, that interpretation is passed to the system in form of outputs that will be read by it [2].

Actuators are devices that produce some behavior in response to an event. When a specific event occurs, there is usually a Sensor that captures and interprets that information. The data is then delivered to the system, which must produce an action triggered by it. That action is done by the Actuator and can be anything, from turning the lights off to increase the heater temperature [2].

With these two types of devices it is possible to define HA scenarios, simply by connecting them. The Figure 17 shows how a Sensor and an Actuator can be connected producing a simple scenario. This example has a Presence Sensor and an Actuator represented by a light bulb. When they are connected, we have a case that when someone enters the room, the light bulb turns on and it turns off if that person leaves.

Figure 17 – Example of a direct connection between a Sensor and an Actuator

5.1.3 Personalize a behavior

The previous example shows two of the (physical) elements used in HA. However, those elements do not have the necessary expressiveness to define more complex behaviors. With this in mind, there is the need to have another type of element called Actions. With this new module, it is possible to enhance a simple behavior like the one previously mentioned.

Actions can be defined as an element that is placed between a Sensor and an Actuator and their objective is to offer more options to refine HA behaviors. Considering the case shown in Figure 17, if the user does not want to turn the light on at its full intensity and wishes to use it at 70% for example, he could do so by using an Action that receives the signal from the Presence Sensor and if there is presence in the room, the Action gives the order to turn the light on at the specified value.

The Figure 18 represents the way to connect an Action to the other elements and how the behavior defined earlier evolved when an Action is used.

Figure 18 – Example of a personalized control of an Actuator

5.1.4 Managing conflicts

The introduction of Actions offers even more flexibility to the user, since this element provides a way to create and improve the behaviors defined. However, there is a problem when one has several Sensors in the same area and those Sensors are competing against the same Actuator. When this happens, the result cannot be predictable because there is not a verification order for each competing behavior. This means that, to manage all those situations, a new type element must be created. These elements are the Decisors, modules that should be placed right before the Actuators and receive several behaviors defined by the combination of Sensors and Actions. To manage all the behaviors connected to a Decisor, a priority system was created. The user should connect all the desired behaviors do the Decisor and then he must define the order in which each one of them will be verified.

This new element is what makes this DSL more adapted to end users than the HAbitATION (Section 3.3.2.2), because with Decisors there is no need to have experts to handle overlapping of behaviors, since it is the user that defines the order by which the actions are evaluated.

The Figure 19 will be used to better understand how Decisors work. As the previous ones, this image is an evolution of the general case. The behavior that keeps the light bulb at 70% is maintained, but now there is another behavior that activates the light bulb if it is night time. Those two behaviors are combined with a Sequential Decisor that evaluates each of them in order and if both conditions are active, then the light is turned on with the value of the first connection, in this case 70%.

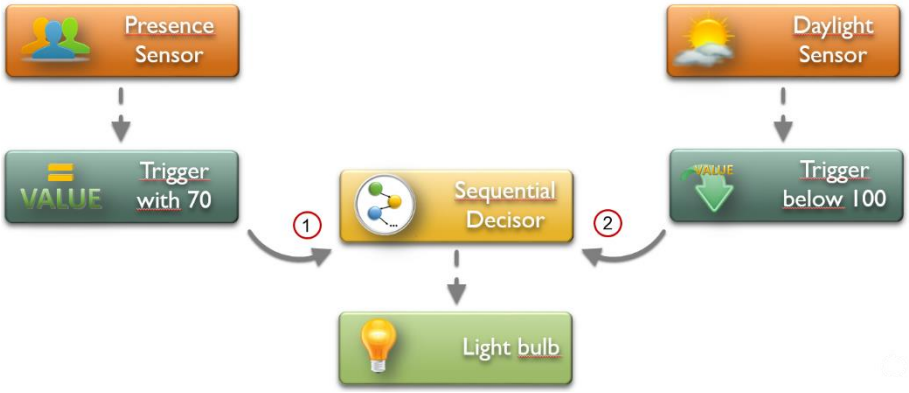


Figure 19 – Example of the usage of a Decisor

5.1.5 Model definition

5.1.5.1 Feature Model

This model expresses the characteristics that are mandatory and optional and also the relationships between them, making it essential in showing the variety in behavior creation.

As shown in Section 9.4.1 (Figure 48), a behavior can include all the elements mentioned before: Sensors, Actuators, Actions and Decisors. Sensors have a type, which can be Presence, Daylight, Temperature and Push Button. All these different Sensors must have an ID that identifies them.

Actuators are passive equipment that are meant to be controlled in some way. There are four types of Actuators: Blinds, Heater and Lights that support dimmer function or that only support the states on and off.

To connect Sensors to Actuators, there are Actions and Decisors. The first ones are modules with logic associated that execute data transformations. This way, the user has the flexibility to manipulate the values, creating a wide diversity of behaviors. There are different types of Actions: activation of something if the input value exceeds a threshold (Trigger Below/Above X), output of a chosen value (Trigger With X), transformation of a value in percentage (Direct/Inverse Percentage) and also increment of a value depending of a time stamp (Step Until X). Decisors are treated as modules that decide between two or more behaviors in three distinct ways: by Priority, where the behavior with more priority is considered if it is active; by Sequence, which means that behaviors are analyzed in order; by Join, where the first behavior to be detected is the one that is executed.

The defined behavior is also composed by two additionally characteristics, Scheduling and Deployment. The Scheduling, allows the scenario to be scheduled to run in specific days or periods, while the Deployment indicates the platform where it will be executed.

5.1.5.2 Domain Model

The domain model represents a conceptual model that describes the solution for this domain, in other words, how the DSL was defined. As represented in Section 9.4.2 (Figure 49), a behavior can be composed by several Sensors, Actions, Decisors and Actuators. Since this model represents the definition of the DSL, there are two other elements that should be considered: Container and Connections.

The first element is where the scenario will be implemented. All the other elements can only exist inside a Container, except the Connections, because a Container supports the way the behavior will be deployed and also a schedule that restricts the execution of the behavior to a date.

The second element are the Connections, which purpose is to connect the elements altogether. There are two types of Connections: Connections to Decisors to connect elements to Decisors since they support order and Standard Connections to connect all the elements except an element to a Decisor.

With all these modules, there are an extensive range of automation behaviors that can be defined. However, there are some limitations when one is making combinations between them. To ensure that a behavior is correctly defined, a set of EVL rules was established as a complement to this model.

5.2 DESIGN

The design phase is composed by the features that will be implemented using a concrete approach. The following topics will cover the DSL's metamodel, the concrete syntax used and the well-formedness rules that compose the language.

5.2.1 Ecore and EMF Models

These two models have a straight correlation with the models described in Section 5.1.5, since they are a textual representation of them. With this, it is possible to define all the DSL elements, their attributes, type and icons, which are the components that constitute the environment.

The diagram generated from these two models is located in Section 9.5, Figure 50 and the EMF file can be found in Section 9.6. The following list further explains the purpose of each class:

- Component – It is the main object of the hierarchy;
- Connection – It is divided into two differente connections that can be used within the Container: DecisorConnection to connect Elements to Decisors and StandardConnection to connect Elements in general;
- DecisorConnection – Contains the Priority attribute, which defines the connection priority;
- StandardConnection – Contains the caseType attribute, which is used with some Elements like the PresenceSensor, to indicate if there is presence or not;
- Container – It is the element that will contain the behavior;
- DomoticModel – The object that makes the connections between the Container's objects, like the Container itself, ContainerConnection and ContainerOperation;
- ContainerConnection – Represents the connection between the Container and the elements that are exterior to it;
- ContainerOperation – Represents the operation that could be executed from the container, like the Deployment type or the behavior Scheduling;
- Deployment – It is divided into two types of deployment, Simulation and Execution;
- Schedule – It is the object that affects the whole behavior. The HourStart attribute is responsible for fixing the starting hour of the behavior and the Duration represents how long the behavior will occur. The Operation is divided in several comparison operations and is used to specify when the behavior will occur. The Days attribute indicate the days when the behavior should be active;
- Element – Is is the class that holds the four main elements used to build a case: Sensor, Actuator, Action and Decisor;
- Sensor – It is the class representation of the physical Sensor. It is divided in PushButton, DaylightSesnor, TemperatureSensor, PresenceSensor and TimeSensor. The PresenceSensor has a Timeout attribute that represents the delay until the actual sensor is not active anymore. The TimeSensor is used for limiting certain actions between the InitialTime and the EndTime;
- Actuator – It is the class representation of the physical Actuator. It is divided into Heater, Blinds, Lock and two types of LightBulb, OnOff and Dimmer;
- Action – Used for introduce more expressiveness to the behaviors. It is composed by TriggerBelowX, TriggerAboveX, TriggerWithX, StepUntilX, DirectPercentage and InversePercentage;

- Decisor – It is the class that holds the three types of decisors that are responsible for avoiding conflict situations. They are the PriorityDecisor, SequentialDecisor and the JoinDecisor. The PriorityDecisor only supports two different actions and the one that has more priority is executed; the Sequential decisor supports several actions, but to execute the action with the ConnectionNumber, the others must be active (works like an AND gate); the JoinDecisor chooses the first action that reaches the decisor without a specific order (works like an OR gate).

5.2.2 Concrete syntax

To create behaviors using a graphical DSL, a set of easily recognizable icons that represent the concrete syntax was defined. Visual notations have extreme importance when the user is using a language that deals with terms of a specific domain, so, for the icons present in the DSL the Daniel Moody's rules were followed [45][46].

To summarize, the user has to recognize without problems the function assigned to a specific icon. Since each person is different and interprets symbols in different ways, all the icons were carefully chosen to have the same meaning regardless who is using the DSL. To assess if the choices for the concrete syntax are suitable, in the usability studies performed (Section 6.2) there is a question dedicated to the evaluation of the icons and a field to leave suggestions. With this type of question, it will be possible to categorize the icons in three types [45]:

- Semantically immediate – If the function of the icon is immediately recognizable from its appearance alone;
- Semantically opaque – If the relationship between the icon and its meaning is purely arbitrary (the icon is acceptable for the function it represents);
- Semantically perverse – If the icon is unrelated or have a different meaning of the function it represents.

Taking into account the way the icons were chosen and all the different functions supported by the DSL, the Table 4 has a representation of the icon, its meaning and also its inputs and outputs when applicable.

Table 4 – DSL icons with descriptions and how they work

Palette Group	*Icon ID*	*Icon*	*Icon Name*	*Icon Description*	*Input/Output*
Object	1		Container	Used to build the domotic behavior	-
	2		Execution	Indicates that the behavior will be executed in the target platform	-
	3		Schedule	A date can be defined for the behavior to run	-
	4		Simulation	Indicates that the behavior will be simulated in the target platform	-

Palette Group	*Icon ID*	*Icon*	*Icon Name*	*Icon Description*	*Input/Output*
Connection	5		Container Connection	Connects the Container to the other Objects	-
	6		Connection to Decisors	Connects elements to Decisors. The order of the connections must begin in 1 with increments of 1 unit	-
	7		Global Connection	Connects the elements: Sensors, Actions and Actuators	-
Sensors	8		Daylight Sensor	Measures and provides the light intensity in lux	Output: The value of the daylight intensity in lux
	9		Temperature Sensor	Measures and provides the temperature in degrees Celsius	Output: The value of the temperature
	10		Presence Sensor	Indicates if there is presence in the room, through movement	Output: True if the presence is detected; False otherwise
	11		Push Button	When pushed it activates/deactivates a behavior. If pressed, the value is given in increments of 10 units	Output: True if it was pushed; False otherwise
	12		Time Sensor	Used to limit the usage of certain Sensors between two distinct hours	Output: True if the time is within the interval defined; False otherwise
Actuators	13		Heater	A simple Heater that could be turned On or Off	Input: A value different of 0 to turn it on
	14		On/Off	A type of light bulb that could only be turned On or Off	Input: A value different of 0 to turn it on
	15		Dimmer	A type of light bulb that supports light intensity variation	Input: A value between 0 and 100.
	16		Blinds	Blinds that are motorized and work with percentage, to define the openness level	Input: A value between 0 and 100.
	17		Lock	A type of lock that can be Locked or Unlocked	Input: A value different of 0 to unlock
Actions	18		Trigger Below X	Indicates that the input value is below the defined Value	Input: An integer value Output: A boolean value
	19		Trigger Above X	Indicates that the input value is above the defined Value	Input: An integer value Output: A boolean value

Palette Group	*Icon ID*	*Icon*	*Icon Name*	*Icon Description*	*Input/Output*
Actions	20		Trigger With X	The output is the defined Value when activated, otherwise has the value 0	Input: A boolean value Output: An integer value
	21		Step Until X	The received base value is modified Step-by-Step after a certain Time, until it reaches the Target value	Input: An integer value Output: An integer value
	22		Direct Percentage	The Value corresponds to 100%, so the output is the direct percentage representation of the input	Input: An integer value Output: An integer value between 0 and 100
	23		Inverse Percentage	The Value corresponds to 0%, so the output is the inverse percentage representation of the input	Input: An integer value Output: An integer value between 0 and 100
Decisors	24		Priority Decisor	Receives 2 connections, where the highest priority overrules the other	Input: Any type of value Output: Any type of value
	25		Sequential Decisor	Supports multiple connections, but only executes an action if the others are active (AND gate)	Input: Any type of value Output: Any type of value
	26		Join Decisor	Supports multiple connections, where the action executed is the first to arrive (OR gate)	Input: Any type of value Output: Any type of value

5.2.3 Well-formedness Rules

There are several well-formedness rules, like connections between the elements that compose a behavior that must be verified before the code is generated. This step is vital to obtain a code without errors, because there are constructions that could generate incorrect situations. Together with the behavior's construction, it is also important to verify if the elements have their mandatory properties filled. To make all this verifications, there is an EVL file that validates the scenario before it is deployed.

All the elements that support a name and/or ID fields should have them filled, because these are properties that identify the element. For Containers, the EVL file confirms aspects like if it does not have another Container inside, if the minimum of elements to define a behavior are present and if all the elements are connected. This last verification is also applied to the elements themselves, as a double check. The Actuators should only support an element connected, while the Decisors only have one output and can only have Connections to Decisors as input. The Connections have to be checked as well, to assess if each Connection is linked to the correct element.

The following list contains all the rules used for each component of the DSL.

Container

- The Container must have a name;
- The Container's name must be a single word, with only letters and/or numbers;
- A Container cannot have a container inside;
- A Container must have at least one Sensor and one Actuator;
- A Container cannot have an isolated behavior;
- A Container must have a Deployment type;
- A Container only supports a Deployment type;
- A Container only supports a Schedule.

Sensor

- The Sensor must have a name;
- The Sensor's name must be a single word, with only letters and/or numbers;
- The name of each Sensor must be different.

Actuator

- The Actuator must have a name;
- The Actuator's name must be a single word, with only letters and/or numbers;
- The name of each Actuator must be different;
- An Actuator only supports one behavior connected to it.

Decisor

- A Decisor can only have one output connection;
- A Decisor can only connect to an Actuator;
- Only a DecisorConnection can be connected to a Decisor.

Priority Decisor

- A Priority Decisor can only have two input connections.

Sequential Decisor

- The property ConnectionNumber must be valid.

StandardConnection

- A StandardConnection cannot have a Decisor as output;
- This type of Sensor does not support Negative behavior;
- The Sensor is not compatible with this Action;
- The Action is not compatible with this Action;
- This element is not compatible with an Actuator.

DecisorConnection

- A Connection to a Decisor must follow a specific order, beginning at 1, with increments of 1 unit;
- A DecisorConnection should have a Decisor as output.

Element

- An Element must have a connection with other

5.3 IMPLEMENTATION

The implemented DSL is based on the models mentioned on the previous sections. This means that those models cannot have construction errors and flaws that could represent a problem in the implementation phase. To avoid it, there was the need to choose a platform that could be reliable and also support all the necessary aspects of the DSL.

As seen in Section 4.2, Epsilon was the chosen tool, because this platform is composed by several tools and languages: the GMF editor generates a front-end based on an Ecore model, modeled using the concepts of HA; the EVL language validates the models with rules using the Ecore elements; the EGL language generates code that will run on the devices or will be used to simulation. To further detail this process, it is necessary to explain how the tools of Epsilon support the DSL.

5.3.1 Ecore and EMF Models

As stated in Section 5.2.1, both of these models describe the metamodel concepts, rules and properties. The Listing 1 represents the specification of the Action StepUntilX of the metamodel (see Section 9.6 for the full EMF file). Since it is an Action, the class extends from the superclass Action. There are four attributes: a readonly ActionType that has the type of the Action (in this case, “Step Unitl X”); a Target that represents the desired target value; the Time indicates the refreshing time to sum the Step to the input received. There is an annotation with a @gmf.node, since this component will be used by the user of the DSL.

Listing 1 – Definition of the StepUntilX class on EMF

```
@gmf.node(label="ActionType", label.icon="true",
       tool.small.bundle="Thesis_Sample_v3.edit",
       tool.small.path="/icons/full/obj16/StepUntilX_24.gif")
class StepUntilX extends Action {
  readonly attr String ActionType = "Step Until X";
  attr int Target;
  attr int Time;
  attr int Step;
}
```

The Figure 20 represents the same Action in the Ecore model (see Section 9.5 for the full Ecore diagram).

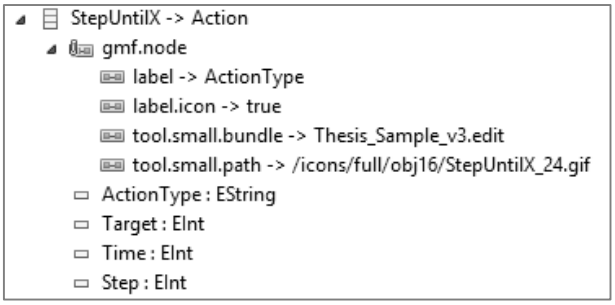


Figure 20 – Definition of the StepUntilX class on Ecore

5.3.2 Well-formedness Rules

As seen in Section 5.2.3, there are several rules defined to verify a behavior. These rules will help the user with the validation of his case step by step. The Listing 2 has the example of the EVL rule for the Container (see Section 9.7 for the full EVL file), which states that a Container cannot have

another Container inside. The rules have a check that verifies the occurrence of the situation, a message that appears and indicates the error and a fix (that is optional) and suggests a way to solve the problem.

In this case, the check verifies if there is a Container inside another and shows the appropriate message. The user can choose to use the recommended fix or solve the problem himself. In the Listing 2, the fix will remove the Container that is inside the other.

Listing 2 – The checkContainer EVL rule of the Container component

```
context Container {
   constraint checkContainer {
      check : not self.hasComponents.exists(t|t.isKindOf(Container))
      message : 'A Container cannot have a container inside'
      fix {
         title : 'Removing Container'
         do {
            for(p in self.hasComponents.select(t|t.isKindOf(Container))){
               delete p;
            }
         }
      }
}}
```

5.3.3 Code Generation

The code generation has two distinct targets, since this DSL is prepared to deploy code for both simulation and execution. It is necessary to study how the target platforms receive code, so that the behavior defined can be deployed correctly.

5.3.3.1 Simulation

The Figure 21 has a general scheme of the connection between the EGL files.

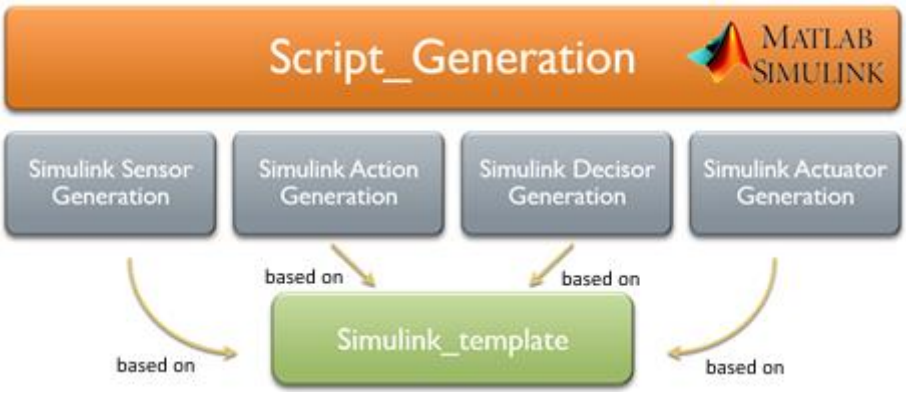


Figure 21 – Scheme of the EGL files for simulation

Simulink_template.egl – This is an auxiliary file that contains several operations that will be called by the other files (Section 9.8.1, Listing 9).

- *initBlockDiagram* – Initiates the block diagram that will represent the simulation context and also the real-time functions;
- *initChart* – Responsible for initialize the chart inside the block diagram;

- *createVariables* – Receives collections of inputs, outputs and chart's internal variables and connects them to the chart itself;
- *statesCreation* – Creates the states inside the chart. This operation uses internally the *defineStatesPositions*, which arranges the states to be in a visible position;
- *transitionsCreation* – Creates transitions between each one of the states;
- *addDefaultState* – Indicates which one is the default state;
- *defineStatesPositions* – It is an operation used internally to place the states in visible positions for the user.

Simulink_Sensor_Generation.egl – It contains the required operations to the implementation of each Sensor. There are Sensors that are more basic than others, which implies the usage of a simple switch to simulate them. Others need supplementary charts to implement more complex behaviors. This file uses the Simulink_templates.egl to build each Sensor.

Simulink_Actuator_Generation.egl – As the previous file, this one has the operations required for the implementation of each Actuator. Since the Actuators are implemented using pre-defined objects of the Simulink library, their representation could be sometimes abstract (Section 9.8.1, Listing 10).

Simulink_Action_Generation.egl – Like the previous files, this one has the required operations for the implementation of each Action. The file Simulink_templates.egl is used for defining the logic associated with each Action.

Simulink_Decisor_Generation.egl – Since the Decisors must have some logic associated, there is also an operation for each Decisor. This file also uses the Simulink_templates.egl to build each Decisor.

Script_Generation.egl – This is the main file that joins together the other files and generates the script file that will be consumed by Simulink. This file starts to identify the different elements that compose a case, creates each one of them and places them in the block diagram of the Simulink. After that positioning, all the connections between the elements are made and after that the simulation is ready. For the purpose of reuse some of the Actions, all of them are encapsulated into a module.

To better understand how the code generation for Simulink works, the Figure 22 has a representation of a simple scenario using the DomATIC and in the Figure 23 the same case in Simulink is shown (Section 9.8.1, Listing 11).

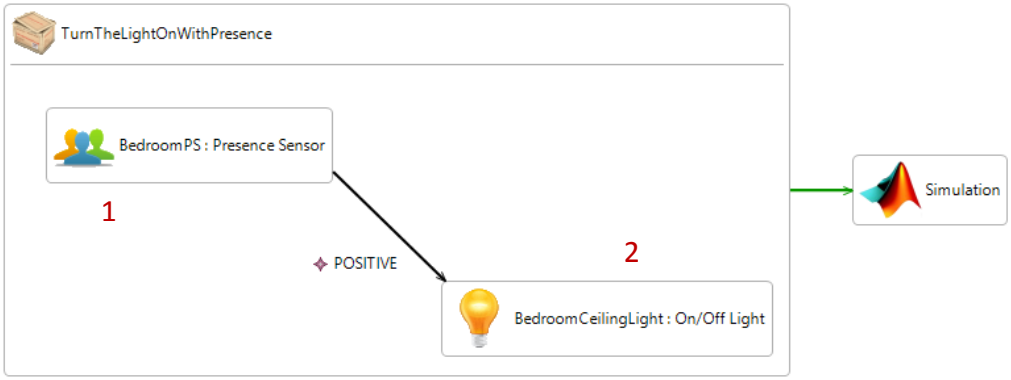


Figure 22 – Turn the light on with presence in DomATIC

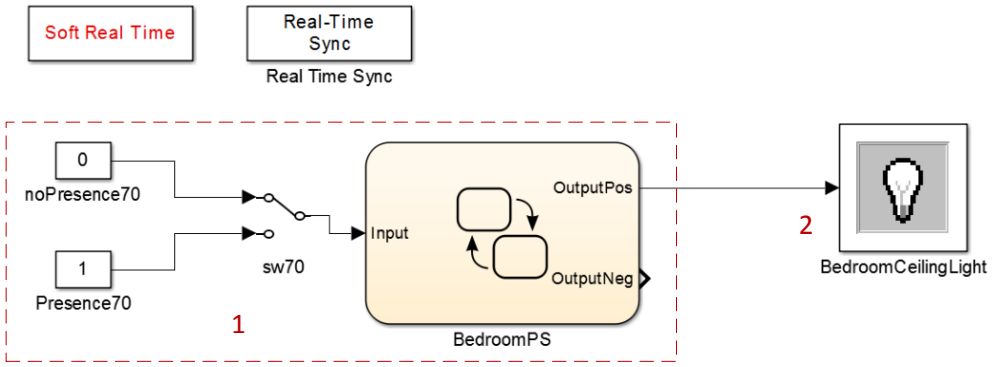


Figure 23 – Turn the light on with presence in Simulink

The number 1 corresponds to the representation of the Presence Sensor and the number 2 to the representation of the Actuator. For the generation of the code of this example, the following pieces of code are needed:

Simulink_templates.egl

(See Section 9.8.1, Listing 9)

Simulink_Actuator_Generation.egl

Listing 3 – The piece of code that generates the Light bulb

```
[% //On_Off Light Bulb
operation onOffLightBulb(actuatorName:String, posX:Integer, posY:Integer)  { %]
%Output block creation
add_block('gauges_gmslib/On Off Gauges/Light Bulb',[sys '/[%=actuatorName%]']);
set_param([sys '/[%=actuatorName%]'],'Position', [[%=posX%] [%=posY%] [%=posX%]+70
[%=posY%]+70]);
[% } %]
```

Simulink_Sensor_Generation.egl

Listing 4 – The code necessary to generate the Presence Sensor

```
[%operation presenceSensorSimulation(sensorName:String, posX:Integer,
posY:Integer, timeout:Integer) { %]
 %Input block creation
 x = [%=posX%];
 y = [%=posY%];
 w = 30;
 h = 20;
 offset = 60;
 press = add_block('Simulink/Commonly Used Blocks/Constant',[sys
 '/noPresence[%=posY%]']);
 set_param([sys '/noPresence[%=posY%]'],'Position', [x y x+w y+h]);
 set_param([sys '/noPresence[%=posY%]'],'Value', '0');
 notPress = add_block('Simulink/Commonly Used Blocks/Constant',[sys
 '/Presence[%=posY%]']);
 y=y+offset;
 set_param([sys '/Presence[%=posY%]'],'Position', [x y x+w y+h]);
 set_param([sys '/Presence[%=posY%]'],'Value', '1');

 switch1 = add_block('Simulink/Signal Routing/Manual Switch',[sys
 '/sw[%=posY%]']);
 x = x+offset+50;
 y = y-(offset/2)-10;
 h = h+20;
 set_param([sys '/sw[%=posY%]'],'Position', [x y x+w y+h]);
 add_line(blockDiagram,'noPresence[%=posY%]/1','sw[%=posY%]/1','autorouting','on'
 );
 add_line(blockDiagram,'Presence[%=posY%]/1','sw[%=posY%]/2','autorouting','on');

 [%var chartNumberRoot = 200;%]
 [%=negativeAndPositiveChart(sensorName, chartNumberRoot, timeout)%]
 set_param([sys '/[%=sensorName%]'],'Position', [200 70 200+150 70+100]);
 add_line(blockDiagram,'sw[%=posY%]/1','[%=sensorName%]/1','autorouting','on');
[% } %]

[%operation negativeAndPositiveChart(chart:String, chartNumber:Integer,
timeout:Integer) {
 var chartName = chart;
 var inputs = Sequence{"Input"};
 var outputs = Sequence{"OutputPos","OutputNeg"};
 var internals = Sequence{};
 var states = Sequence{"Init","NoValue","Value"};
 var statesPosition = Sequence{Sequence{1,0},Sequence{0,1},Sequence{2,1}};
 var statesInnerInstructions = Sequence{"","du:OutputNeg = 1\\nexit: OutputNeg =
 0","du:OutputPos = 1\\nexit: OutputPos = 0"};
 var statesConnections = Sequence{Sequence{1,2},Sequence{0},Sequence{0}};
 var statesConnectionsPosition = Sequence{Sequence{Sequence{6,0},Sequence{6,0}},
 Sequence{Sequence{9,9}}, Sequence{Sequence{3,3}}};
 var statesConnectionsLabels = Sequence{
  Sequence{"[Input == 0]","[Input == 1]"},
  Sequence{"after(" + timeout + ",sec) [Input == 1]"},
  Sequence{"[Input == 0]"}
 };%]

 [%=initChart(chartName, chartNumber)%]
 [%=createVariables(inputs, outputs, internals, chartNumber)%]
 [%=statesCreation(states, statesConnections, statesInnerInstructions,
 statesPosition, chartNumber)%]
 [%=transitionsCreation(states, statesConnections, statesConnectionsPosition,
 statesConnectionsLabels, chartNumber)%]
 [%=addDefaultState(states.first(), chartNumber)%]
[% } %]
```

Generated script

Listing 5 – The script generated for Simulink
(does not correspond to the example)

```
sfnew
rt = sfroot
m = rt.find('-isa','Simulink.BlockDiagram')
ch = m.find('-isa','Stateflow.Chart')

sA = Stateflow.State(ch);
sA.Name = 'A';
sA.Position = [50 50 310 200];
sA1 = Stateflow.State(ch);
sA1.Name = 'A1';
sA1.Position = [80 120 90 60];
sA2 = Stateflow.State(ch);
sA2.Name = 'A2';
sA2.Position = [240 120 90 60];
tA1A2 = Stateflow.Transition(ch);
tA1A2.Source = sA1;
tA1A2.Destination = sA2;
tA1A2.SourceOClock = 3;
tA1A2.DestinationOClock = 9;
tA1A2.LabelPosition = [180 140 0 0];
tA1A2.LabelString = 'E1';
pos = tA1A2.LabelPosition;
pos(1) = pos(1)+5;
tA1A2.LabelPosition = pos;

% Add a default transition to state A
dtA = Stateflow.Transition(ch);
dtA.Destination = sA;
dtA.DestinationOClock = 0;
xsource = sA.Position(1)+sA.Position(3)/2;
ysource = sA.Position(2)-30;
dtA.SourceEndPoint = [xsource ysource];
dtA.MidPoint = [xsource ysource+15];
% Add a default transition to state A1
dtA1 = Stateflow.Transition(ch);
dtA1.Destination = sA1;
dtA1.DestinationOClock = 0;
xsource = sA1.Position(1)+sA1.Position(3)/2;
ysource = sA1.Position(2)-30;
dtA1.SourceEndPoint = [xsource ysource];
dtA1.MidPoint = [xsource ysource+15];

sfsave(m.Name, 'myModel');
```

5.3.3.2 *Execution*

The connection between the EGL files are very similar to the previous one and that is represented in Figure 24 .

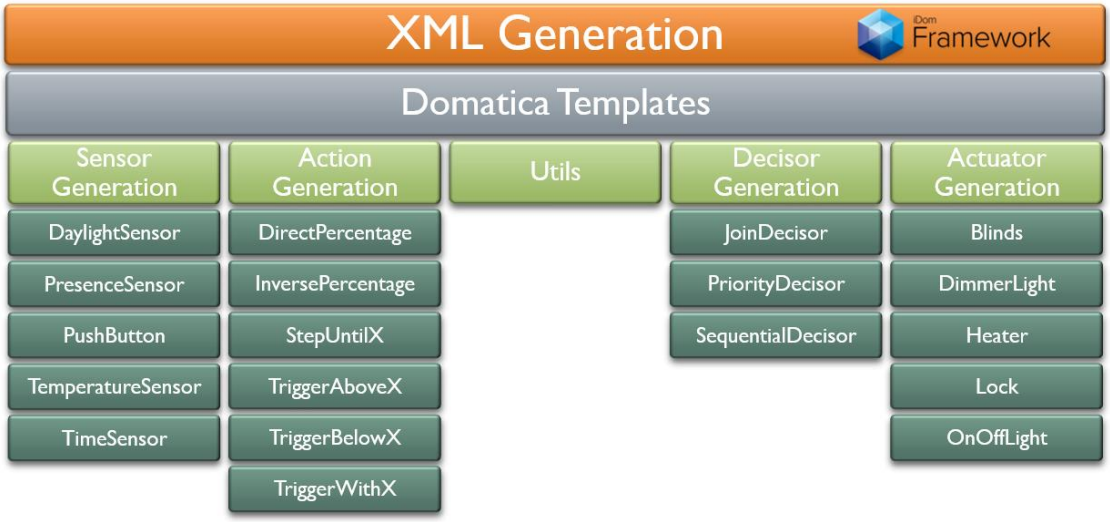


Figure 24 – Scheme of the EGL files for execution

Each Sensor, Actuator, Action and Decisor have a file that handle their specific characteristics. All these files follow the same structure as the main file of each element.

Domatica_Sensor_Generation.egl – Responsible for the creation of the Sensors' user parameters, initial state, user tasks, events, verifications and values (Section 9.8.2, Listing 12).

Domatica_Actuator_Generation.egl – Responsible for the creation of the Actuators' initial state and user tasks.

Domatica_Action_Generation.egl – Responsible for the creation of the Actions' user parameters, initial state, user tasks, events, verifications, values and variables.

Domatica_Decisor_Generation.egl – Responsible for the creation of the Decisors' user parameters, initial state, user tasks, events, verifications and values.

Domatica_Calendar_Generation.egl – The Calendar must be treated separately, because the Domatica device handles the time using its internal system. This file has the operations that insert lines of code for the time and date into the user tasks for each type of component.

Domatica_Utils.egl – Creates the default user tasks needed to start a case.

Domatica_templates.egl – Responsible for calling all the above files' operations. It begins with the default user tasks of Domatica_Utils and then goes to the other files and calls their operations depending the behavior built on the DSL (Section 9.8.2, Listing 13).

XML_Generation.egl – Executes the operations present on the Domatica_templates file (Section 9.8.2, Listing 14).

All the decisions about each one of the specific elements are handled inside each one of the main files. The operations are always called and the decision is made by a switch statement.

To better understand how the code generation for iDom Framework is generated, the Figure 25 represents a simple scenario using the DomATIC and in the Listing 6 the resulting XML file is shown.

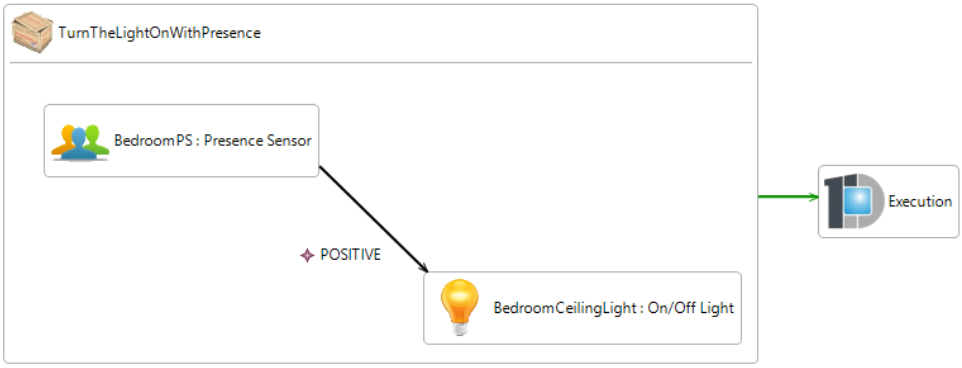


Figure 25 – Turn the light on with presence in DomATIC

Listing 6 – The XML code generated for iDom Framework

```
<?xml version="1.0" encoding="UTF-8"?>
<iDomUserProgram>
  <UserParameters>
    <UserParameter id="#FCT#GCOUNTER" idname="GCounter" name="GCounter" type="VARIABLE"
    enabled="true"/>
    <UserParameter id="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS"
    idname="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS"
    name="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS" type="VARIABLE"
    enabled="true"/>
    <UserParameter id="#FCT#Presence_Check#TurnTheLightOnWithPresence#BedroomPS"
    idname="#FCT#Presence_Check#TurnTheLightOnWithPresence#BedroomPS"
    name="#FCT#Presence_Check#TurnTheLightOnWithPresence#BedroomPS" type="VARIABLE"
    enabled="true"/>
    <UserParameter id="#FCT#Presence_Value#TurnTheLightOnWithPresence#BedroomPS"
    idname="#FCT#Presence_Value#TurnTheLightOnWithPresence#BedroomPS"
    name="#FCT#Presence_Value#TurnTheLightOnWithPresence#BedroomPS" type="VARIABLE"
    enabled="true"/>
  </UserParameters>
  <UserTasks>
    <UserTask id="#FCT#TSK#GCounter" idname="#FCT#TaskGlobalCounter"
    name="GlobalCounterIncrement" enabled="true" isprocess="false">
      <Events> <On op1="SystemClock" operation="" op2=""/> </Events>
      <Code> <Add op1="#FCT#GCOUNTER" op2="1"/> </Code>
    </UserTask>
  <UserTask id="#FCT#TSK#Init" idname="#FCT#TaskInitialize" name="Task Initialize" enabled="true">
      <Events> <On op1="$Start" operation="" op2=""/> </Events>
      <Code>
        <Assign op1="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS" op2="0"/>
        <Assign op1="#FCT#Presence_Check#TurnTheLightOnWithPresence#BedroomPS" op2="0"/>
        <Assign op1="#FCT#Presence_Value#TurnTheLightOnWithPresence#BedroomPS" op2="0"/>
        <Assign op1="DEV00000007.control" op2="0"/>
      </Code>
  </UserTask>
```

```
    <UserTask id="#FCT#TSK#TurnTheLightOnWithPresence#BedroomPS"
    idname="#FCT#TSK#TurnTheLightOnWithPresence#BedroomPS"
    name="TurnTheLightOnWithPresence#BedroomPS" enabled="true">
      <Events> <On op1="DEV00000004.detect" operation="EQUAL" op2="1"/> </Events>
      <Code>
        <Assign op1="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS" op2="0"/>
        <Assign op1="#FCT#Presence_Value#TurnTheLightOnWithPresence#BedroomPS" op2="100"/>
        <Assign op1="#FCT#Presence_Check#TurnTheLightOnWithPresence#BedroomPS" op2="1"/>
      </Code>
    </UserTask>
    <UserTask id="#FCT#TSK#TurnTheLightOnWithPresence#BedroomPS_Negative"
    idname="#FCT#TSK#TurnTheLightOnWithPresence#BedroomPS_Negative"
    name="TurnTheLightOnWithPresence#BedroomPS_Negative" enabled="true">
      <Events>      <On op1="DEV00000004.detect" operation="EQUAL" op2="0"/> </Events>
      <Code>
        <If op1="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS" operation="GREATER"
        op2="0"/>
        <Return/>
        <End/>
        <Assign op1="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS"
        op2="#FCT#GCOUNTER"/>
        <Add op1="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS" op2="5"/>
      </Code>
    </UserTask>
    <UserTask id="#FCT#TSK#Presence_WT_End#TurnTheLightOnWithPresence#BedroomPS"
    idname="#FCT#TSK#Presence_WT_End#TurnTheLightOnWithPresence#BedroomPS" name="Verify if
    Presence_WT has passed" enabled="true">
      <Events> <On op1="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS"
      operation="MINOROREQUAL" op2="#FCT#GCOUNTER"/> </Events>
      <Code>
        <If op1="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS" operation="EQUAL"
        op2="0"/>
        <Return/>
        <End/>
        <Assign op1="#FCT#Presence_WT#TurnTheLightOnWithPresence#BedroomPS" op2="0"/>
        <Assign op1="#FCT#Presence_Value#TurnTheLightOnWithPresence#BedroomPS" op2="0"/>
        <Assign op1="#FCT#Presence_Check#TurnTheLightOnWithPresence#BedroomPS" op2="0"/>
      </Code>
    </UserTask>
    <UserTask id="#FCT#TSK#OnOff#TurnTheLightOnWithPresence#BedroomCeilingLight"
    idname="#FCT#TSK#OnOff#TurnTheLightOnWithPresence#BedroomCeilingLight"
    name="#FCT#TSK#OnOff#TurnTheLightOnWithPresence#BedroomCeilingLight" enabled="true">
      <Events>
        <On op1="#FCT#Presence_Check#TurnTheLightOnWithPresence#BedroomPS" operation="EQUAL"
        op2="1"/>
        <On op1="#FCT#Presence_Check#TurnTheLightOnWithPresence#BedroomPS" operation="EQUAL"
        op2="0"/>
        <On op1="DEV00000007.control" operation="NOTEQUAL"
        op2="#FCT#Presence_Value#TurnTheLightOnWithPresence#BedroomPS"/>
      </Events>
      <Code><Assign op1="DEV00000007.control"
      op2="#FCT#Presence_Value#TurnTheLightOnWithPresence#BedroomPS"/>
      </Code>
    </UserTask>
  </UserTasks>
</iDomUserProgram>
```

5.3.4 Deployment

This DSL allows the behavior to be exported for two platforms, one for simulation and other for execution with real devices. The simulation platform allows the behavior to be simulated before it is sent to the devices, validating its consistency. This type of operation is most useful for the domain experts, because they can perceive the flow and interaction between elements.

With the intent to validate the DSL with real devices, the constructed behavior can also be transformed into runnable code that is suitable to be interpreted by programmers, allowing them to improve it if necessary.

5.3.5 Tool layout

The Figure 26 shows the layout of the tool, which was used in the usability tests of the Section 6.2.

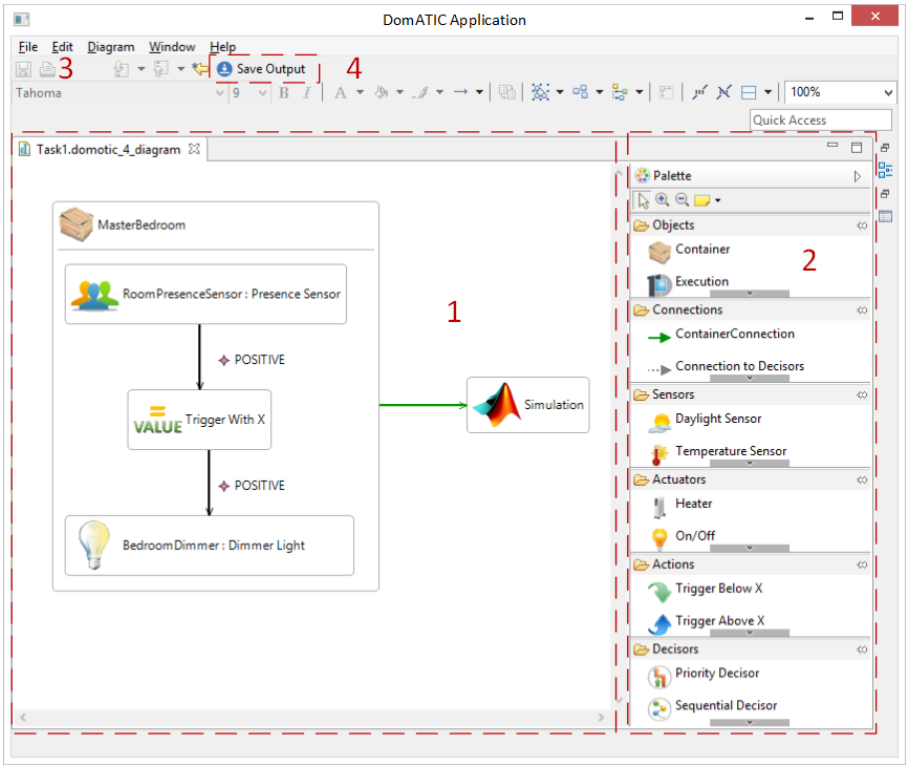


Figure 26 – Layout of DomATIC

1. This is the area available for the user to build his behavior;
2. The palette where the user can pick the several elements of the DSL;
3. In the Edit menu, the user has the option to Validate the behavior, which will aplly the well-formedness rules of the EVL file;
4. Saves the output (Simulink script or Domatica xml) to the chosen location.

6

LANGUAGE VALIDATION

This last DSL construction phase consists in verifying the work developed in the previous ones, which was detailed in Chapter 5. It is necessary to test it with real users to determine if the language is adapted to their demands. To verify this, it is necessary to prepare a set of usability cases that will be presented to them and will contribute to assess if there are aspects of the DSL that should be improved.

There is, however, another way to strengthen the validation of the proposed language. Considering that the HAbitATION DSL is the most similar, it is reasonable to compare it to the DomATIC framework. With this comparison, we can confirm if our language is in fact adapted to domain experts and if it solves some of the problems discussed by Manuel Buendía, like the overlapping effect mentioned in Section 3.3.2.2.

6.1 COVERED CONCERNS

In Section 3.3.2.3, we explored some of the concerns that DSLs, like Monaco and HAbitATION, should cover. We conclude that two of the main concerns that were not covered were the behavior simulation and the inexistence of well-formedness rules to validate if the behavior is built without errors.

Table 5 recovers those concerns, but this time they are applied to DomATIC DSL. Alongside the previous concerns, there are two more characteristics that we found important to be present in a DSL for this domain, which are the behavior encapsulation and the equivalence between the simulation and execution models.

Table 5 – Concerns of Home Automation covered by DomATIC in comparison with the previous DSLs (● - Full implementation; ◑ - Partial implementation; ○- Not supported)

Concerns	*Monaco*	*HAbitATION*	*DomATIC*
HA concepts	●	●	●
Model reuse	◑	●	●
Graphical notations	◑	●	●
End-user oriented	○	●	●
Well-formedness rules validation	○	○	●
Behavior simulation	○	○	●
Behavior deployment	●	●	●
Behavior encapsulation	○	○	○
Simulation and execution models equivalence	○	○	○

The behavior encapsulation is a feature that is not present in this version of the DSL, because there are some limitations of Epsilon. To implement this feature, it would be necessary to have much more experience with Epsilon that would require to study the tool more, which could hamper us to follow other paths. So, for this version, DomATIC is focused in solving the flaws of the other DSLs for HA and this feature as well as other suggestions from the usability tests, will be implemented later.

The equivalence between the simulation and the execution models is a feature that should be implemented later. For this phase the equivalence is only guaranteed with the assurance that both models follow the same logic of implementation, but there is not a concrete test that assess this correspondence.

6.2 USABILITY STUDIES

The last step to validate the DomATIC language is to conduct evaluation tests with real users to assess if the DSL is ready to be used in real-life situations. For this evaluation, the focus must be not only on non-technical people, but also on domain experts/programmers. To achieve this, the experience was conducted using people from these two groups, with the following usability goals and objectives in mind [47]:

- Comprehension – The user should understand if the described behavior is possible to accomplish using the tool;
- Readability – The user should identify what a certain behavior means;
- Problem Solving – The user should perform a series of tasks to evaluate if he can easily use the tool;
- Usefulness – The user is asked about his willingness to use a tool like this for the purpose it was designed.

Alongside these objectives, there are other metrics that will be used to analyze the performance of the testers. The ISO 9241-11 states that usability is "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [48]. The effectiveness will be measured through the correctness shown by the testers when building a task. The efficiency is achieved by the time the user took to perform the task. The degree of satisfaction is taken through the opinions of the participants about the language.

This experience has another goal, which is to identify the flaws of the DSL. This is done using the opinion expressed by the users, but also by observing the users' performance during the task and the questions they asked. At this point, it is significant to have the opinion of the users about the icon choices, because icons that are not appropriate can complicate the usage of the tool.

6.2.1 Conducting the experience

For this experience, there is the need to have people with different degrees of knowledge. Since it is difficult to distinguish programmers from domain experts in this domain, for this experience the users were divided into Novices, Domain Experts and DSL Users.

The experiences were conducted in *Faculdade de Ciência e Tecnologia* (FCT) and in *Instituto Superior Técnico* (IST) with the participation of 12 students, with a duration about 60 minutes. From this group, 5 students were considered Novices, since they their area of knowledge is mathematical models. Other 5 students were considered Domain Experts because of their background in this area of study and their proficiency in using HA devices. The last 2 students have background knowledge in DSLs, so they were considered DSL Users. The Table 6 summarizes this information, as well as other profiling data.

Table 6 – Participants' profile information

User ID	*Group*	*Age*	*Sex*	*Country*	*Education Level*	*Field of Study*
1	Novice	21	F	Poland	Secondary	System Engineering
2	Novice	20	M	Poland	Secondary	System Engineering
3	Novice	21	M	Poland	Secondary	System Engineering
4	Novice	21	F	Poland	Secondary	System Engineering
5	Novice	24	F	Poland	Master	System Engineering
6	Domain Experts	24	M	Portugal	Bachelor	Computer Science
7	Domain Experts	25	M	Portugal	Master	Computer Science
8	Domain Experts	26	M	Portugal	Master	Embedded Systems
9	Domain Experts	25	M	Portugal	Bachelor	Master in Robotic
10	Domain Experts	22	M	Portugal	Bachelor	Computer Science
11	DSL Users	24	M	Portugal	Bachelor	Computer Science
12	DSL Users	24	M	Portugal	Bachelor	Computer Science

For the purpose of this test, there were some questions made regarding the experience that the participants had with equipment and software of this domain.

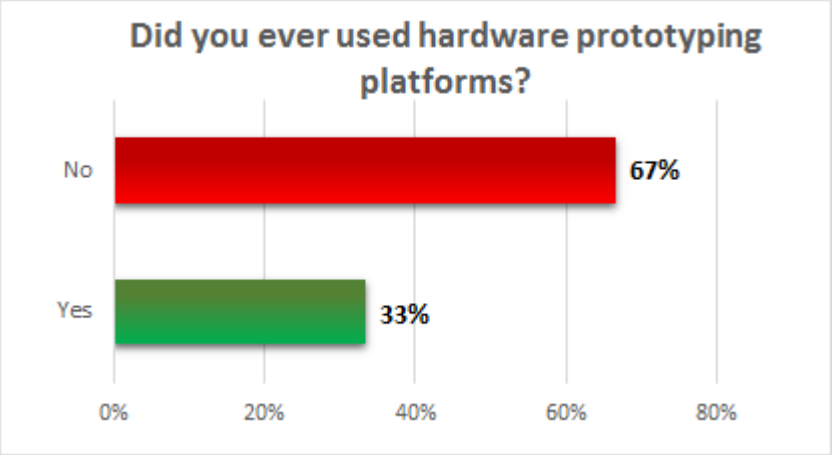


Figure 27 – Answers to the hardware prototyping experience of the users

Regarding the prototyping platforms just like Arduino and RaspberryPI (Figure 27), only 33% of the users had ever use them. Two of the responses came from the group of Domain Experts, while the other two came from the DSL Users. None of the Novices had experience with this type of devices.

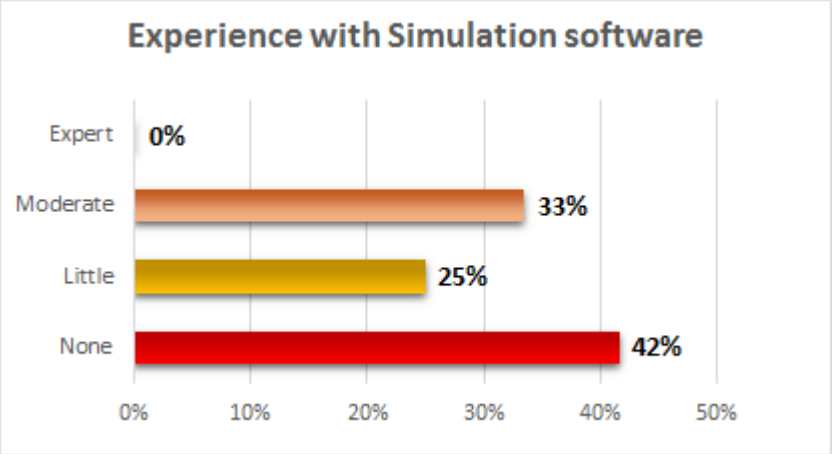


Figure 28 – Answers about the experience with Simulation software

Examining Figure 28, we found that none of the users is an expert in this type of software, but 58% (33% + 25%) of them at least had contact or worked often with a Simulation tool. The Novices and the Domain Experts are the groups that take part in the 58%.

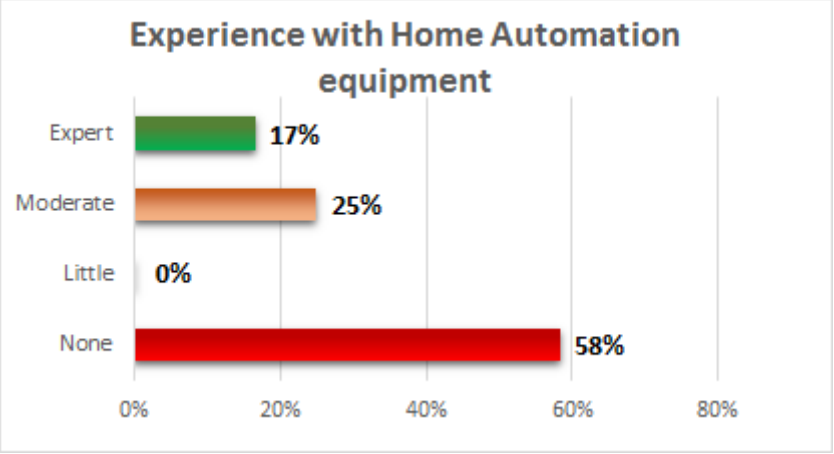


Figure 29 – Answers about the experience with Home Automation equipment

As we expected, only the group of Domain Experts participants are experts or have a moderate experience with this type of equipment (Figure 29). All of the five answers were from this group, while the other seven participants from the other groups never handled this type of devices.

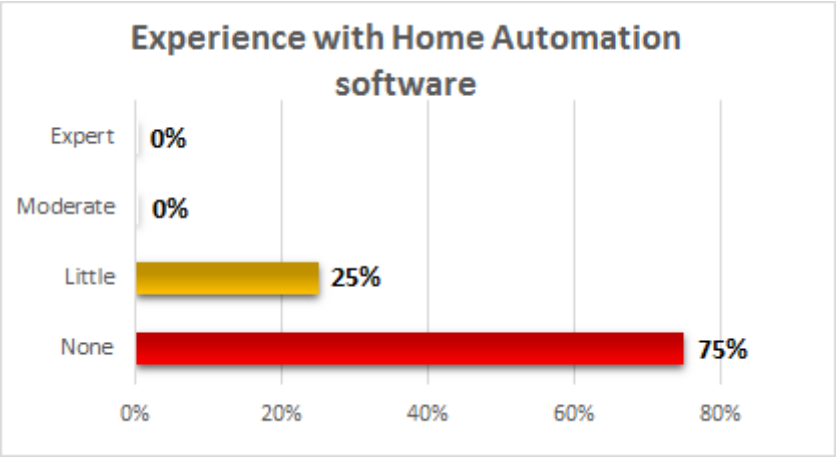


Figure 30 – Answers about the experience with Home Automation software

From Figure 30, it is possible to observe that only 25% of the participants have little experience with HA software and all the others have none experience. Those 25% are part of the Domain Experts group, which means that they have contact with the hardware and not with the software.

From all the above graphics, it is possible to conclude that Domain Experts are the group that have more experience with the equipment and software used in this domain. However, it is interesting to note that they do not have much practice with the software that controls the devices, perhaps because it is too difficult to understand or because it is proprietary and it is not available.

6.2.2 Experience procedure

There was a total of three sessions for each one of the groups. The first and third tests were conducted at FCT, while the other was performed at IST.

The experience procedure was the same for the three groups, so we could have results the more accurate as possible. The materials used for this experience were the following:

- One computer for each participant;
- A questionnaire (divided in two parts);
- A set of slides for an explanation of the language;
- Recording software (to register the actions performed by the participants) ;
- The DSL and auxiliary files.

The Figure 31 has the general process used for the DSL evaluation.

Figure 31 – General process for the DSL evaluation

The first step was to ensure that all the participants' computers were ready to run the DomATIC language, because there could be some problems with the versions of Java. Then, the recording software was installed to analyze the steps of each participant. When the setup was completed, the Part I of the questionnaire (see Section 9.10) was given to the testers. After they finished the Part I, there was a brief explanation about the language using the slides previously done. To finalize the experiment, the participants were asked to respond to the Part II of the questionnaire. To show all the DSL features, the participants executed their tasks in the Simulink environment and in the Domatica equipment. With this, they could see in real-time how a behavior can be simulated and then deployed to be performed in a real-life situation.

6.2.2.1 Presentation

A set of presentation slides were prepared to be showed between the Part I and II of the questionnaire, which is significant to give an explanation about the DSL. The slides are divided as follows:

- Examples with text of some possible behaviors that could be done using the language. Those behaviors could go from the simple "turn the light on when there is presence in the room" to "turn the ventilation system on when I am parking the car, but only for 20 minutes and if the CO_2 level is above a certain threshold";
- A section where all the components of the DSL are explained one by one;
- A DSL example section, with behaviors built using the tool. These examples are accompanied with text to ensure that the testers associate the expressions to the respective component.

6.2.2.2 Questionnaire

As previously stated, the questionnaire is divided in two parts (Section 9.10). The Part I is the first contact that the tester has with the language and contains four main sections:

1. General Information – This is where the user provides some of his personal data, like the age, sex and country;
2. Education – This section is used to determine what is the actual education level of the participant and also what is his field of study;
3. Experience – Since this DSL is meant to be used in an HA domain, it is significant to know about the experience of the user with hardware prototyping platforms, simulation software, home automation equipment and home automation software.

4. Icons Validation – In this section it is asked to the user to give his opinion about the icons chosen for the DSL. In the questionnaire, a table is presented with the icon, its name and its description, and the user must select which type of association is more adequate to each icon, according to the Moody's rules (Section 5.2.2).

It is extremely important that the Part I is the first contact of the user with the language, because that way he is not biased by previous knowledge that he could have acquired.

Part II is composed by six sections that are meant to evaluate the participant's performance, which follows the usability goals and objectives mentioned in the beginning of this chapter. Those sections are:

1. Tasks to perform – The user is challenged to use the DSL to create behaviors. There are three tasks proposed and each of them can be rated in difficulty and accomplishment. This section has the objective of evaluate the Problem Solving feature;
2. Automatic Behaviors – The participant must answer two questions about the capacity of the language in implementing those behaviors. The goal is to assess if the user Comprehends the extension of the language;
3. Behavior identification – In this section, two images representing two behaviors are provided. The user should describe what each image represents so we could evaluate his Readability with the language;
4. General Opinion – We ask the questioned person what is the task that was most difficult to execute and if he recommends this tool for the purpose it was designed. This is used to check the Usefulness of the language. As a final question in this section, we want some suggestions of the participant about the icons. This question is asked after all of the tests, so we could have an opinion of someone who had contact with the DSL and is aware of what it does;
5. Additional Feedback – The goal is to have a little more feedback of the user about what is well implemented and what could be improved;
6. Personal Information – We give the opportunity for the participant to fill his personal information if he wants to be contacted in the future about the evolution of this project.

6.2.3 Results

In this section, the results obtained from all the experiences will be shown. Each sub-section represents one of the evaluation points that were explained in the chapter introduction.

6.2.3.1 Concrete syntax evaluation

As previously mentioned, we asked the participants to categorize the icons into three categories (Section 5.2.2):

- Immediate Association (Semantically immediate) – Means that the icon is immediately recognizable and there is no doubt about its function on the DSL;
- Logical Association (Semantically opaque) – The relation between the icon and the function is acceptable;

- No Association (Semantically perverse) – The icon and the function are completely unrelated.

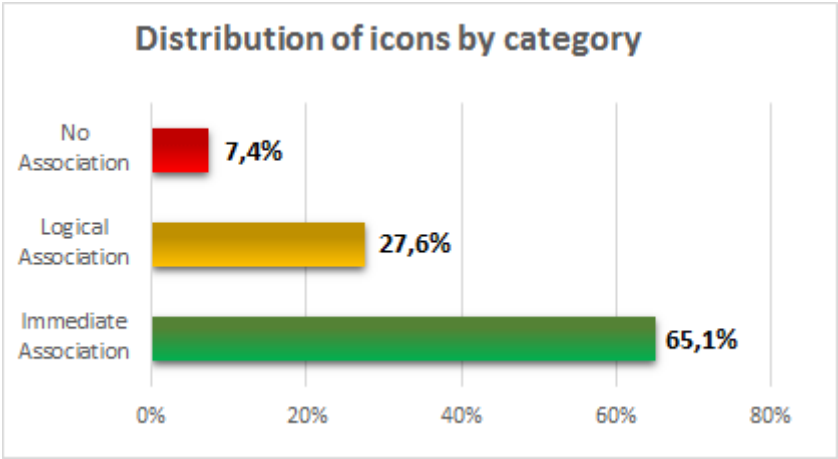


Figure 32 – Graphic that represents the distribution of icons by category

The Figure 32 summarizes the overall results obtained for all the icons. From this graphic, it is possible to conclude that the participants considered that the icons were carefully chosen, because 65% stated that there was an Immediate Association. In terms of Logical Association, 28% of the responses considered the icons appropriate, however there are 7% of the responses that did not like the icons at all.

With a total of 12 participants and 26 icons to choose, it is essential to identify which are the icons that can be improved.

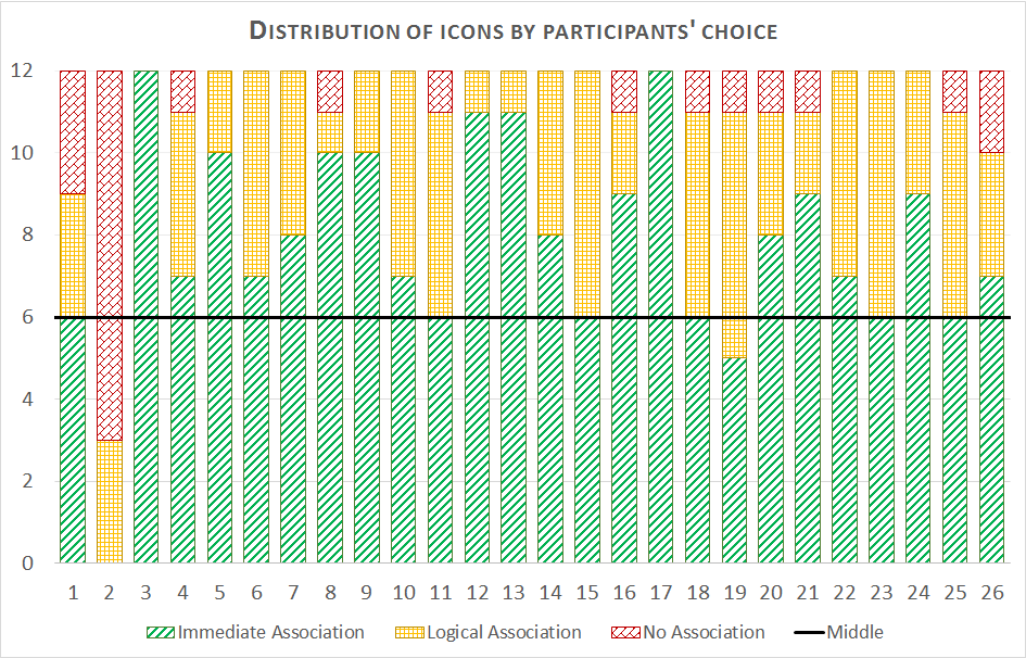


Figure 33 – Participants' choices by icon

From Figure 33 it is possible to identify which are the icons that were considered poorly chosen. If we only consider the icons below the Middle line, the only two choices are the icons number 2 and 19, since the others are on the line or above. Since we want to improve the DSL as much as we can, we have considered also the icons that are on line, which means that the participants

are not quite sure about them as well. With this, now the icons number 1, 11, 15, 18, 23 and 25 were considered too.

Table 7 – The concrete syntax icons considered poorly chosen

Icon ID	*Icon*	*Icon Name*	*Icon Description*
1		Container	Used to build the domotic behavior
2		Execution	Indicates that the behavior will be executed in the target platform
11		Push Button	When pushed it activates/deactivates a behavior. If pressed, the value is given in increments of 10 units
15		Dimmer	A type of light bulb that supports light intensity variation
18		Trigger Below X	Indicates that the input value is below the defined Value
19		Trigger Above X	Indicates that the input value is above the defined Value
23		Inverse Percentage	The Value corresponds to 0%, so the output is the inverse percentage representation of the input
25		Sequential Decisor	Supports multiple connections, which are validated by priority

The Table 7 has the actual representation of these icons. The suggestions about how the icon should look are listed below:

- Icon 1 – Two suggestions were made to change this icon into a script file or a folder to provide a more sense of encapsulation;
- Icon 2 – This icon is tricky since it represents the platform of execution and the participants did not have that in mind. The users preferred to have an icon with the words “Play” or “Go”, and also something like sprockets;
- Icon 11 – The users found this icon hard to visualize. The suggestions were a more obvious switch or a finger pushing the button;
- Icon 15 – The users wanted the dimmer to have something like a scale to indicate that the light level is variable;
- Icons 18 and 19 – There were almost no suggestions about these icons, since they are hard to express using an image. We thought that they could be more perceptible with a line as a threshold and the words above/below in the image;
- Icons 23 – This is another complicated icon since the concept is difficult to express. There was no suggestions for this one and we do not have another idea for the icon, which means that more tests should be done to assess if it is a good or bad choice;

- Icon 25 – There were no suggestions either for this icon. The conclusion is the same as icon 23, because we and half of the participants considered this icon as a good one.

6.2.3.2 Comprehension of the tool

In the Part II of the questionnaire, the section Automatic Behaviors is meant to evaluate this feature. The user has two behaviors described by text and must identify if they can be accomplished using this DSL.

Table 8 – Answers about the capacity of the tool to perform two distinct behaviors

User ID	Behavior 1	Behavior 2
1	✓	✓
2	✓	✓
3	✗	✓
4	✓	✓
5	✓	✓
6	✓	✓
7	✓	✓
8	✓	✓
9	✓	✓
10	✓	✓
11	✓	✓
12	✓	✓
Total	91,67%	100%

The two behaviors were possible to define using this DSL, and as we can see in Table 8, the majority of users considered that as well. There is only one exception in Behavior 1, because one of the participants did not identify that the behavior was possible to perform, but he did not provide any reason why he considered that.

6.2.3.3 Readability

The section Behavior identification in the Part II of the questionnaire was used to identify if before a behavior the participant is aware of what it does. There were two distinct behaviors that the users had to identify and explain what their result was.

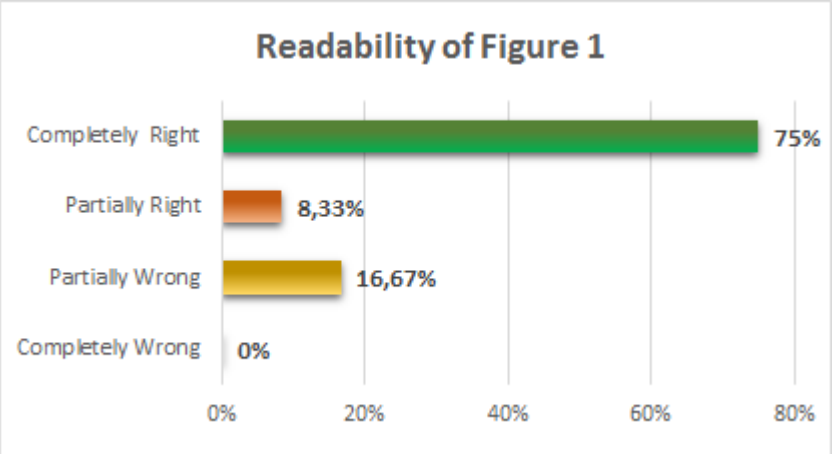


Figure 34 – Graphic that represents the readability results of Figure 1 of the questionnaire

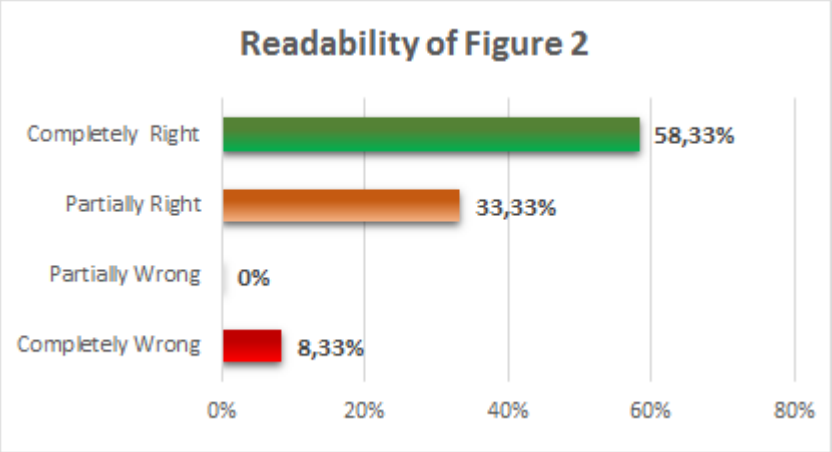


Figure 35 – Graphic that represents the readability results of Figure 2 of the questionnaire

For both behaviors, the majority of users identified what was being asked. Looking at Figure 34, a third of the users have identified correctly the case. The remaining participants did not identified what the case meant, but they did not failed to identify the behavior completely. In Figure 35, it is possible to verify that almost 60% of the users have identified the case correctly and about 33% of the participants have identified it partially right. However, this behavior was more difficult to identify, since 8,33% of the testers have failed in the identification.

6.2.3.4 Problem solving

In the section "Tasks to perform" in the Part II of the questionnaire, the testers had to build three behaviors using the tool. To analyze the performance of the users, in this section it makes sense to use some of the usability metrics (effectiveness and efficiency), to assess if the group achieved a satisfactory usage of the tool.

There are two indicators that should be used for this measurement, which are the Mean and the Standard Deviation. The first is used to have an indication of the mean time the user takes to execute each task. This is important to distinguish the users that are efficient from the ones that are effective. The second indicator is used to measure if the difference between each users' performance times differ significantly from each other.

Table 9 – Summary of the results obtained from each task

	Mean time	*Standard Deviation*	*Efficient users*	*% of efficient users*	*Effective users*	*% of effective users*
Task 1	8,5	3,86	5	41,67%	9	75,00%
Task 2	6,93	1,74	3	25,00%	9	75,00%
Task 3	12,46	4,85	3	25,00%	8	66,67%

The Table 9 summarizes the results obtained from each task for the group of 12 participants. The tasks were established to be increasing in complexity and it is possible to verify that the mean time decreases between tasks 1 and 2, but increases between tasks 2 and 3. This means that from the task 1 to task 2 there is some learnability about the language, since the difference between them lies in the quantity of elements. The difference from task 2 to task 3 is more about thinking than execution, which explains the time that task 3 consumed. This fact is also explained with the standard deviation of each task. The task 1 is the first time that the participants used the tool, making the differences between users more noticeable. The same is

applied to task 3, but in terms of thought. For the task 2, the value is considerably lower, due to nature of repetition of the task.

From Table 9 it is possible to take other conclusions. If we compare the percentage of efficient users and effective users, there is a significant gap between both values. This fact is more noticeable for tasks 2 and 3, where the participants had performed the tasks correctly, but not as fast as expected. From this data and the feedback of the participants, we determined that some of the DSL components are difficult to understand and to manipulate, which is restraining the testers' speed when building the case.

6.2.4 Threats to validity

In every evaluation process there are some threats that undermine the test result in some way. This section will be used to identify the possible limitations and influences that may comprise the experience.

The three sessions of evaluation were executed using the process described in Section 6.2.2, but the conditions of the room, the time of the day and even the nationality of the participants were different:

- The first session occurred at FCT, in a room designated only for testing the DSL. The experience was conducted after lunch and we must be aware that the testers could be less focused due to the time of the day it occurred. Other factor is that the 5 users were from Poland, which means that some the explanation was made in English and some extra explanation for some components was required.
- The second session occurred at IST and once again, the room was destined only to the DSL testing. This session happened before lunch and the concentration of the users was considerable greater than the users of the first session. The other advantage was the language, since for this session the communication was made in Portuguese, which raised more questions and more feedback about the language.
- The third session was performed at FCT, but the room were not exclusively reserved for the experience, which led to some distraction occurrences. This session happened after lunch, but the participants were focused most of the time and performed above the expectations and conditions of the room.

Despite all these threats, we have tried to execute the sessions as equivalent as possible. We think that effort provided reliable results that were not significantly influenced by the conditions described, but we have to take into consideration all the conditions used in each one of the sessions.

6.2.5 Discussion

The results obtained throughtout this chapter and the comments and feedback provided by the participants, allowed us to identify some of the problems of the language.

As can be verified in Table 10, the DSL Users executed all the tasks correctly, followed by the Novices group, which is the target group of the DSL. With the group of Domain Experts, the experience did not have the results expected, because of the overthinking showed by this group when they were performing the tasks. Most of them wanted to understand in detail how the DSL worked, and that fact contributed to some misperception about some of the elements. However, this was the group that identified several flaws in the tool.

Table 10 – Summary of the tasks execution by group

Groups	*Total of users*	*% of users*	*% of correct answers Task 1*	*% of correct answers Task 2*	*% of correct answers Task 3*
Novices	5	41,67%	80%	100%	60%
Domain Experts	5	41,67%	60%	40%	60%
DSL Users	2	16,67%	100%	100%	100%

In the first iterations of the language, the modules were more closed, where a single module provided the function to detect presence and turn the light on. This first idea was suspended, because it did not offer the required expressiveness needed for this domain and we chose to give the users more autonomy to build their cases in this phase. This idea surfaced again by the Domain Experts suggestion, because they asked if some of the behaviors could be encapsulated into a custom module. This is a concern that will be implemented in the next phase of the language, because we consider it an essential functionality.

The Domain Experts wanted more personalization options and a description about the inputs and outputs for each module, which is understandable given their area of expertise.

The EVL, which is responsible to detect errors in the behavior, proved to be a useful mechanism, since the users were learning with the alerts. In the videos taken from the experience, it is possible to observe the increasing of the participants' knowledge from task to task when they frequently used the EVL's warnings. Since the users were often using this feature, some alerts that were not included initially were identified and will be implemented in the next phase of the language.

The majority of the concrete syntax icons proved to be properly chosen, but there were some suggestions that raised questions about the functionality and the chosen icon. Some of these suggestions will be taken into account for the next usability test of the next phase.

For the next phase, some efforts about the reduction of the complexity of some icons will be made in order to diminish the difference identified between efficient and effective users, since we strive for a DSL where all the users are efficient.

6.3 COMPARISON WITH OTHER APPROACHES

Throughout this document, both Monaco and HAbitATION languages are mentioned a few times as a source of comparison with the DomATIC language. The best aspects of both languages were used as an inspiration and some of their flaws were studied and adjusted to the development the current DSL. We have seen that Monaco is only adaptable to programmers because of its complexity, but the HAbitATION language is adequate towards domain experts and has more abstract visual interface. Since the HAbitATION and DomATIC DSLs share the same characteristics, a comparison between them can be made to evaluate which one introduces scenarios with less complexity, as well as if they support the automation of ordinary circumstances of everyday life.

6.3.1 Case study

As a case study for the evaluation of these two languages, the Figure 36 represents a floor plant which represents a house with its Sensors and Actuators and the Table 11 describes where the elements are placed in the house.

Figure 36 – A floor plant of a habitation with its Sensors and Actuators

Table 11 – Element's positioning through the house

Area ID	Area Name	Element	Element ID	Element Type
1	Garage	Sensor	PS_G	Presence Sensor
		Actuator	SL_G	On/Off Light
2	Hall	Sensor	PS_H	Presence Sensor
		Actuator	SL_H	On/Off Light
3	Living room	Sensor	DLS_LR	Daylight Sensor
			TS_LR	Temperature Sensor
		Actuator	B_LV	Blinds
			H_LR	Heater
4	Kitchen	Sensor	PS_K	Presence Sensor
		Actuator	SL_K	On/Off Light
5	Master bedroom	Sensor	PB_B1	Push Button
			DLS_B1	Daylight Sensor
			PS_B1	Presence Sensor
		Actuator	DL_B1	Dimmer Light

With Figure 36 and Table 11 as basis, a scenario that combines several devices was developed:

"When I enter in my Garage after a day of work, generally between 7 and 8 PM, I want to turn on automatically the lights there, in the Kitchen and in the Hall. If the temperature in the Living Room is below 15°C, I want the Heater to start heating the room. It is usual for me to go straight into the Kitchen, but I do not want to turn off the other lights manually. After dinner, I like to see some

movies in my Living Room that contains many plants, which have the need to absorb as much sunlight as they can. This means that I want to have the blinds in Living Room open depending on the light outside. In the bedroom, I want the lights to be on only when I am there, if it is night time and if it is between the 8am and the 11pm. Seeing that is useless to waste energy I only want the lights to be at 60%. However, I may need more light intensity, so I want the room the be lit at 100% with Push Button. Consequently, I only want this bedroom scenario to work during the working days."

To represent this complex scenario we need to divide it by its different rooms. We will have three sub-scenarios, Garage, Living Room and Bedroom. Each sub-scenario is represented with DomATIC and HAbitATION DSLs. Since we do not have access to the actual HAbitATION language, the implemented scenarios were built based on the description given by the author [36].

Garage

In this sub-scenario the user wants to turn a set of lights on when he enters the Garage. This can be done using a Presence Sensor (PS_G) to detect the opening of the gate. When this sensor is activated, the lights SL_G, SL_H and SL_K are turned on. The user also wants to turn the Heater (H_LR) on if the temperature in the Living Room is below 15°C, which is detected using the Temperature Sensor (TS_LR). To ensure that the lights are turned off automatically, a Presence Sensor is placed in the Hall (PS_H) and in the Kitchen (PS_K) as well. All these actions are programmed only between 7 and 8 PM, which is the interval that the user arrives at home.

This scenario is implemented in HAbitATION (Figure 37) an in DomATIC (Figure 38) languages.

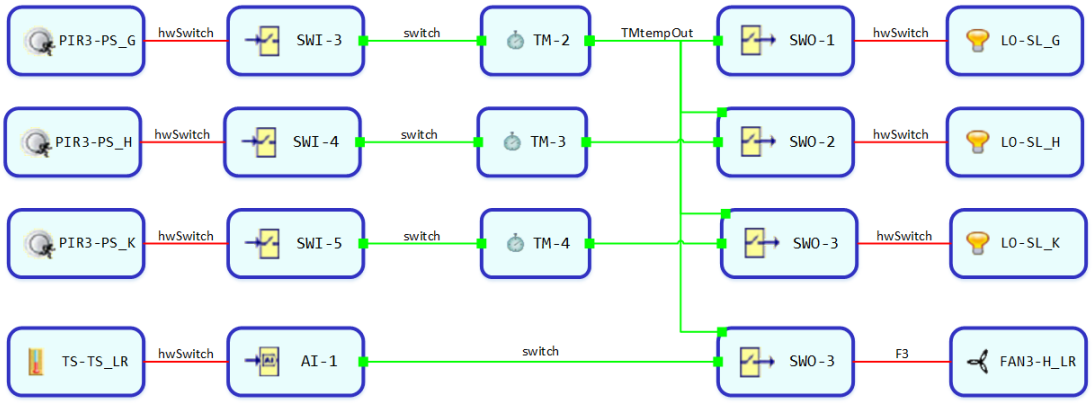


Figure 37 – Garage sub-scenario implemented with HAbitATION

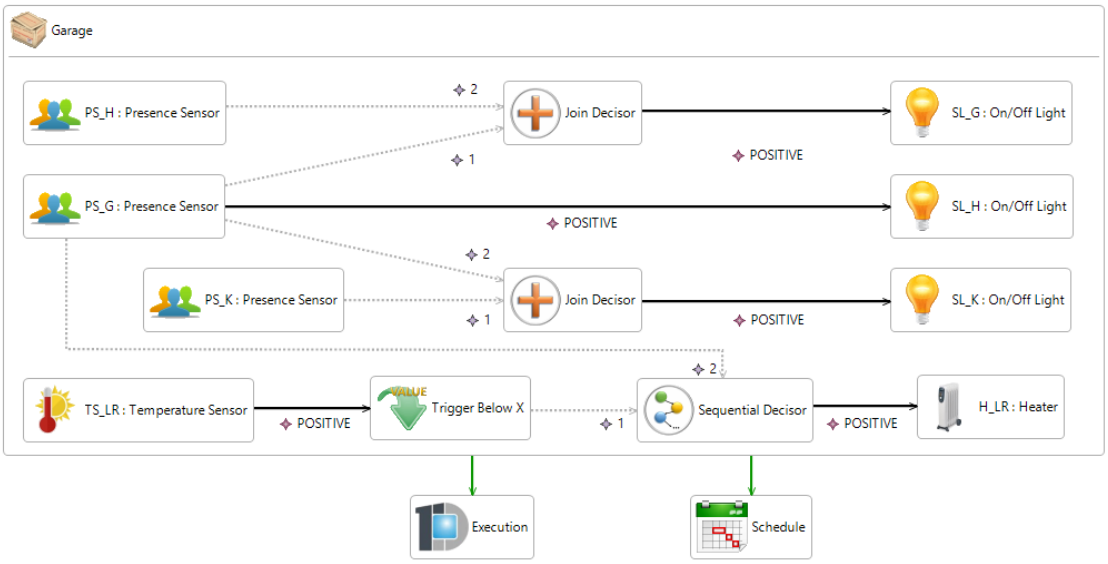


Figure 38 – Garage sub-scenario implemented with DomATIC

Living Room

The sub-scenario in the Living Room is more controlled than the Garage one, because the Sensors and Actuators involved are all in the room. The user wants to control the blinds (B_LR) depending on the light outside. To achieve this behavior, a Daylight Sensor (DLS_LR) is needed to detect the amount of light and open/close the blinds accordingly.

This implementation of this scenario is defined in HAbitATION (Figure 39) and in DomATIC (Figure 40) languages.

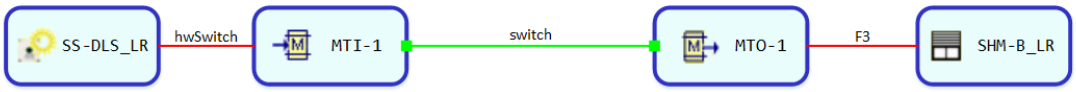


Figure 39 – Living Room sub-scenario implemented with HAbitATION

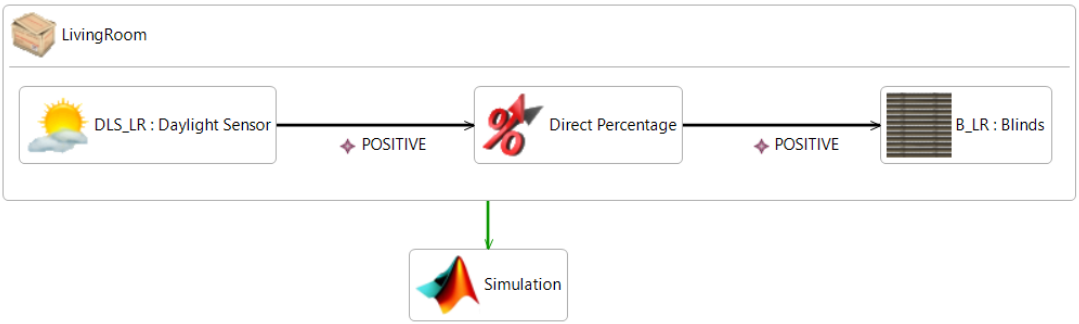


Figure 40 – Living Room sub-scenario implemented with DomATIC

Bedroom

The user wants the lights to be on in the bedroom only if a combination of three things occur: it must be dark outside, there should be someone inside and it only worked between the 8 AM the 11 PM. This is achieved with a Presence Sensor (PS_B1), a Daylight Sensor (DLS_B1) that controls the light in the ceiling (DL_B1) and a Time Sensor. Besides having this automatic behavior, the user

wants to manually control the intensity of the light. For this, he needs a Push Button (PB_B1) to control the light and surpass the other actions.

This scenario definition was modeled in HAbitATION (Figure 41) and in DomATIC (Figure 42) languages.

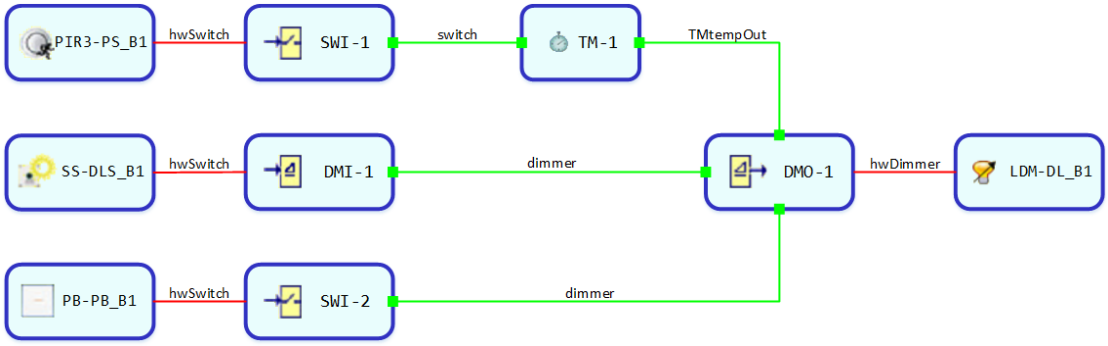


Figure 41 – Bedroom sub-scenario implemented with HAbitATION

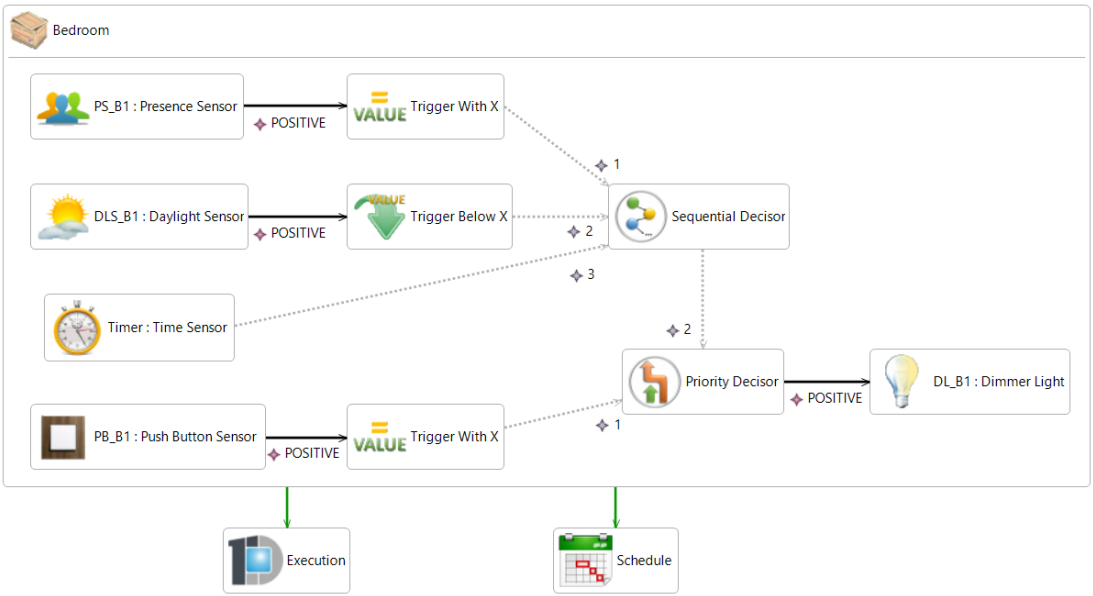


Figure 42 – Bedroom sub-scenario implemented with DomATIC

6.3.2 Comparison between DSLs

When modeling the same scenarios using the two DSLs, it becomes evident that both have some aspects in common, like the elements and their positioning. Reading the scenarios from left to right, we can see that Sensors are the first elements to be placed and Actuators are the last ones, because this is the natural order of placement when the case is being constructed. There are however differences in the logic, which are worth mentioning for comparison purposes.

The first main difference is the overlapping effect mentioned by Manuel Buendía. When the user is creating the scenario, he can connect the several elements but he cannot be sure about the order they will be interpreted and if there are conflict situations, like a Sensor turning a light on and other turning it off. To handle a situation like this, it is not only needed a domain expert but also a DSL expert to fine-tune the case so the overlapping effect does not occur. In DomATIC this effect does not exist because of the Decisors. These elements act as an intelligent filter that receives several

actions and manage them in the order defined by the user. This way, the user do not have to be an expert in the DSL, since giving an order of verification to an element is a natural thing to do.

Another difference is the controllers used in HAbitATION. Each one of the Sensors/Actuators have a controller associated, which can have some parameters to personalize the behavior. These controllers are then connected to other controllers to build the scenario. In DomATIC, there are no controllers, because the approach of this language is to use generic logic to build a case. The problem of this is that not all elements are compatible, leading to incorrect cases. The solution is the EVL file that holds the rules of connection between elements. With this validation file, the user defines a scenario, confirms it and if errors were made, there are suggestions of correction. With this method, we lose complexity and have the advantage of validating the scenario before it is simulated/deployed.

The last difference is the scheduling of a behavior. In our DSL, the case is built inside a Container and the module Calendar can be applied to the whole scenario, as seen in Figure 38 and Figure 42. With HAbitATION, when there is the need to affect the entire construction with a Week Programmer (as the author calls it), there is no clear place to do it. This module can only be connected to certain controllers, which means that to make a general scheduling it should be connected to all the controllers in the scenario.

As a last note, the figures that represent the scenario construction with HAbitATION could not be the correct ones since we do not have access to the actual language. We are sure that the elements used in Figure 37, Figure 39 and Figure 41 have, at least, the minimum elements to build the scenario, but some details such as the activation of an action when the temperature is below 15 ºC may be incorrect or even not possible to achieve. Another functionality that we are not certain that exists, is the scheduling of a scenario as a whole and the restriction of specific actions considering a time period defined by the user.

The mentioned flaws make the HAbitATION DSL unfitting for end users, but also difficult to be used by domain experts. Some of the decisions made in the construction of the DSL, like the fact that some connections must be made with extreme care to avoid overlapping situations, makes this language hard to be used without some amount of time dedicated for explaining how it works. The DomATIC language provides a solution much more simple for both end users and domain experts, but also for programmers who can actually manipulate the code. This solution is more immediate and its construction avoids confusion when the scenarios become more complex.

6.4 DISCUSSION OF THE RESULTS

From both the usability test and the comparison with HAbitATION, we were more aware about the potential of this DSL, but also about its flaws.

Starting with the usability test, most of the negative feedback was directed to the fact that the DSL does not support the encapsulation of behaviors. The users (in particular the group of Domain Experts) wanted to reuse the behaviors between tasks and in this version of the language this is not possible. We consider that this is the natural step of evolution and it will be implemented in the next language iteration.

Another issue was the absence of some information regarding the input and output of the elements, which must be corrected as well. There were no other major problems when the users tested the language, since most of the concrete syntax icons were perceptible and recognizable. The EVL file that

contains the well-formedness rules guided the user when we made errors in the construction of a behavior and this file just need little modifications for the next phase.

Regarding the comparison between our approach and HAbitATION, both DSLs are very similar, but as we mentioned in Section 6.3.2, the HAbitATION seems more complicated to use. The complexity of the behaviors appears to be the same, but the HAbitATION suffers from its dependency from experts in the DSL when there is the need to build a more complex case. The concrete syntax used for this DSL do not follow the rules of usability, but we cannot say if they were chosen correctly, since the author did not conducted usability tests. Finally, the scheduling of behaviors is unclear on HAbitATION, because the author did not give enough information how to do it. From we have determined, it is not possible to schedule a whole behavior unless the module used for it is connected to all the existent controllers, which means a tremendous effort it the case has a lot of these elements.

7

CONCLUSIONS

In this document, we have presented a Domain Specific Language based on the Model Driven Development methodology for Home Automation. The proposed language fills an existing gap in this domain, which is to empower users with an adequate language to express the behavior of their automation systems. There are other languages like Monaco and HAbitATION that represent an abstraction of the system, but their purpose is not to be used by home users not necessarily proficient in programming, since those platforms lack usability features. Despite this, these two languages represent two different perspectives of the domain, which was taken into account in the development of the DomATIC. In fact, the best aspects of those languages contributed as a positive inspiration to produce a language suitable for programmers, domain experts and end users.

The validation of the language occurred in two ways: with a usability test with potential users of the DSL; and by a direct comparison between DomATIC and HAbitATION. The usability tests helped in the identification of some errors in the DSL, but most important was the feedback provided by the participants that will be used in the next phase of the tool. On the other hand, when comparing DomATIC with HAbitATION, we could conclude that our approach is more oriented towards the end users. The language also produces scenario's constructions less confusing that do not need to be fine-tuned by an expert in the DSL. We think that even a domain expert will have more difficulty in using HAbitATION than DomATIC, due to the flaws mentioned throughout this paper.

7.1 CONTRIBUTIONS

The main contribution that comes from this work, is all the process that led to the growth of a tool that defined a new way to look at this domain. The meeting with experts, the procedure of building the language and all the ideas that flourished while DomATIC was being developed, can be a starting point to change the domain of Automation. The DomATIC language has three main aspects: code validation, simulation and deployment. With the implementation of these different perspectives, we developed a language prepared to define Home Automation scenarios that can be verified before they are simulated or deployed, reducing future errors. To increase even further the error reduction, the simulation phase offers an overview of the system's way of operation before the deployment phase, which reduces time and costs.

Another contribution is the systematic analysis that was conducted about the tools similar to this one, which tried to define a way to control devices without much success. Some of them have flaws that are undesirable for a tool that tries to set a new paradigm in this domain. This analysis is enriched with a comparison between the tools and the existent equipment, with both their good and bad characteristics.

The final contribution is the validation process that occurred with real users. This process contributed for the development of our DSL, but the ideas used in the process can be exploited for the validation of other DSLs or similar products. The validation of the concrete syntax and the questionnaire's tasks are the main points of this process we think is simple to follow and contributes with important and vital information for the evolution of the product.

7.2 FUTURE WORK

DomATIC can be improved in many ways, since this first phase had some flaws, most of them appointed in the usability test sessions. One of the natural evolutions of this language is the encapsulation of the behaviors in modules. When a user builds several behaviors and the "light from the garage must turned on when there is presence" is required in all of them, the user obviously wants a single module that does that action. This will be the main concern for the next phase of the evolution of DomATIC. The concrete syntax icons can be subjected to improvements too, since some of them were identified as being confusing and did not had a cognitive connection with the action they should perform. The validation phase for the next language iteration should be performed with more participants, in order to get more accurate results about some of the DSL problems.

Another interesting work, could be on truly guarantying the correspondence between the execution semantics in the platform and the simulation behavior in the simulation environment, to certify the equivalence between both of them.

As a final idea, considering the fact that this language follows a MDD approach, the DSL can be easily expanded to other sub-domains of Automation. With this advantage, we have the potential to explore the automation in factories, university campus, hotels and several other types of buildings.

8

REFERENCES

[1] P. Sánchez, M. Jiménez, F. Rosique, B. Álvarez, and A. Iborra, “A framework for developing home automation systems: From requirements to code,” *J. Syst. Softw.*, vol. 84, no. 6, pp. 1008–1021, Jun. 2011.

[2] P. J. Clemente, J. M. Conejero, J. Hernández, and L. Sánchez, “HAAIS-DSL: DSL to develop home automation and ambient intelligence systems,” in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, 2009, pp. 13–18.

[3] M. Chan, E. Campo, D. Estève, and J.-Y. Fourniols, “Smart homes—current features and future perspectives,” *Maturitas*, vol. 64, no. 2, pp. 90–97, 2009.

[4] M. Jahn, M. Jentsch, C. R. Prause, F. Pramudianto, A. Al-Akkad, and R. Reiners, “The Energy Aware Smart Home,” in *Future Information Technology (FutureTech), 2010 5th International Conference on*, 2010, pp. 1–8.

[5] V. Miori, L. Tarrini, M. Manca, and G. Tolomei, “An open standard solution for domotic interoperability,” *Consum. Electron. IEEE Trans.*, vol. 52, no. 1, pp. 97–103, 2006.

[6] B. Selic, “The pragmatics of model-driven development,” *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003.

[7] B. Appukuttan, T. Clark, S. Reddy, L. Tratt, and R. Venkatesh, “A model driven approach to model transformations.,” in *Workshop on Model Driven Architecture: Foundations and Applications*, 2003, pp. 1–12.

[8] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005.

[9] M. Voelter, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*, vol. 2, no. 65. 2010, p. 556.

[10] E. Seidewitz, “What Models Mean,” *IEEE Softw.*, vol. 20, no. 5, pp. 26–32, 2003.

[11] J. Gray, K. Fisher, and C. Consel, “Panel DSLs : The Good , the Bad, and the Ugly,” vol. 45, 2008.

[12] C. Atkinson and T. Kuhne, “Model-driven development: a metamodeling foundation,” *Software, IEEE*, vol. 20, no. 5, pp. 36–41, 2003.

[13] E. Rios, T. Bozheva, A. Bediaga, and N. Guilloreau, “MDD Maturity Model: A Roadmap for Introducing Model-Driven Development.,” in *ECMDA-FA*, 2006, vol. 4066, pp. 78–89.

[14] M. Wenger, M. Melik-Merkumians, I. Hegny, R. Hametner, and A. Zoitl, “Utilizing IEC 61499 in an MDA control application development approach,” *2011 IEEE Int. Conf. Autom. Sci. Eng.*, pp. 495–500, Aug. 2011.

[15] T. Reus, H. Geers, and A. van Deursen, “Harvesting Software Systems for MDA-Based Reengineering .,” in *ECMDA-FA*, 2006, vol. 4066, pp. 213–225.

[16] J. Pavón, J. J. Gómez-Sanz, and R. Fuentes, “Model Driven Development of Multi-Agent Systems,” in *ECMDA-FA*, 2006, pp. 284–298.

[17] C. Atkinson and T. Kühne, “A generalized notion of platforms for model-driven development,” in *Model-driven software development*, Springer, 2005, pp. 119–136.

[18] K.-H. John and M. Tiegelkamp, *{IEC} 61131-3 : programming industrial automation systems : concepts and programming languages, requirements for programming systems, aids to decision-making tools*. Berlin, Heidelberg: Springer-Verlag, 2001.

[19] N. Völker and B. J. Krämer, “Automated verification of function block-based industrial control systems,” *Sci. Comput. Program.*, vol. 42, no. 1, pp. 101–113, Jan. 2002.

[20] J. X. Mansell, A. Bediaga, R. Vogel, and K. Mantell, “A Process Framework for the Successful Adoption of Model Driven Development.,” in *ECMDA-FA*, 2006, vol. 4066, pp. 90–100.

[21] T. Pacifica, *Easy X10 Projects For Creating A Smart Home*. Indy-Tech Pub, 2005, p. 160.

[22] B.-H. Kim, K.-H. Cho, and K.-S. Park, “Towards LonWorks technology and its applications to automation,” in *Proceedings KORUS 2000. The 4th Korea-Russia International Symposium On Science and Technology*, 2000, vol. 2, pp. 197–202.

[23] D. Loy, D. Dietrich, and S. Hans-Joerg, *Open Control Networks: LonWorks/EIA 709 Technology*. Springer, 2001, p. 368.

[24] M. Ruta, F. Scioscia, E. Di Sciascio, G. Loseto, and E. Di Sciascio, “Semantic-Based Enhancement of ISO/IEC 14543-3 EIB/KNX Standard for Building Automation,” *IEEE Trans. Ind. Informatics*, vol. 7, no. 4, pp. 731–739, Nov. 2011.

[25] K. Thramboulidis, G. Frey, and S. Member, “An MDD process for IEC 61131-based industrial automation systems,” in *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, 2011, pp. 1–8.

[26] E. Tisserant, L. Bessard, and M. de Sousa, “An Open Source IEC 61131-3 Integrated Development Environment,” *2007 5th IEEE Int. Conf. Ind. Informatics*, pp. 183–187, Jul. 2007.

[27] A. Rensink and J. Warmer, *Model Driven Architecture-Foundations and Applications*, no. June. Springer-Verlag, 2006.

[28] J. Han, J. Yun, J. Jang, and K. Park, “User-friendly home automation based on 3D virtual world,” *IEEE Trans. Consum. Electron.*, vol. 56, no. 3, pp. 1843–1847, Aug. 2010.

[29] J. Lapon, K. Vangheluwe, V. Naessens, A. Vorstermans, and K. H. Sint-lieven, “A Generic Architecture for Secure Home Automation Servers,” in *Proceedings of the Third European Conference on the Use of Modern Information and Communication Technologies*, 2008, pp. 261–271.

[30] J. Froehlich, “Promoting energy efficient behaviors in the home through feedback: The role of human-computer interaction,” in *Proc. HCIC Workshop*, 2009, vol. 9.

[31] R. Harper, *Inside the smart home*. London: Springer, 2003.

[32] P. Gabriel, M. Goulão, and V. Amaral, “Do Software Languages Engineers Evaluate their Languages?,” *CoRR*, vol. abs/1109.6, 2011.

[33] A. Bariic, V. Amaral, and M. Goulao, “Usability Evaluation of Domain-Specific Languages,” *2012 Eighth Int. Conf. Qual. Inf. Commun. Technol.*, vol. 0, pp. 342–347, 2012.

[34] M. Voelter and I. Groher, “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development,” *11th Int. Softw. Prod. Line Conf. (SPLC 2007)*, pp. 233–242, Sep. 2007.

[35] H. Prähofer, D. Hurnaus, R. Schatz, C. Wirth, H. Mössenböck, and H. M. Herbert Prähofer, Dominik Hurnaus, Roland Schatz, Christian Wirth, “Monaco: A DSL Approach for Programming Automation Systems.,” *Softw. Eng.*, vol. 121, pp. 242–256, 2008.

[36] M. J. Buendía, “Desarrollo de Sistemas Domóticos utilizando un enfoque dirigido por Modelos”, PhD, Universidad Politécnica de Cartagena, 2009.

[37] F. Hermans, M. Pinzger and A. Van Deursen, “Domain-specific languages in practice: A user study on the success factors,” in *Model Driven Engineering Languages and Systems*, Springer, 2009, pp. 423–437.

[38] D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[39] J. Merseguer and S. Bernardi, “Dependability analysis of DES based on MARTE and UML state machines models,” *Discret. Event Dyn. Syst.*, vol. 22, no. 2, pp. 163–178, Jul. 2011.

[40] C. Klein, C. Prehofer, and B. Rumpe, “Feature Specification and Refinement with State Transition Diagrams,” in *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, 1997.

[41] T. Murata, “Petri nets: Properties, analysis and applications,” *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[42] I. Kurtev, J. Bézivin, F. Jouault, P. Valduriez, and A. Inria, “Model-based DSL frameworks,” in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 602–616.

[43] E. Full, C. Generation, S. Kelly, J.-P. Tolvanen, J. Gray, A. Gokhale, S. Neema, and J. Sprinkle, *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, 2008, p. 444.

[44] J. Travis and J. Kring, *LabVIEW for Everyone: Graphical Programming Made Easy and Fun*. Prentice Hall PTR, 2006.

[45] D. Moody, “The ‘Physics’ of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering,” *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 756–779, Nov. 2009.

[46] D. L. Moody, P. Heymans, and R. Matulevicius, “Improving the Effectiveness of Visual Representations in Requirements Engineering: An Evaluation of i* Visual Syntax,” in *2009 17th IEEE International Requirements Engineering Conference*, 2009, pp. 171–180.

[47] J. Rubin, *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[48] T. Jokela, N. Iivari, J. Matero, and M. Karukka, “The Standard of User-centered Design and the Standard Definition of Usability: Analyzing ISO 13407 Against ISO 9241-11,” in *Proceedings of the Latin American Conference on Human-computer Interaction*, 2003, pp. 53–60.

9

APPENDIX

This chapter contains some extra information relative to the others, like tables and images that in a way do not fit in the flow of the general document.

9.1 MEETINGS WITH DOMAIN EXPERTS

Several meetings with domain experts have taken place, in order to assess if the following conclusions about this domain were relevant for the developing of this DSL.

9.1.1 Home Automation typical scenarios

The following four scenarios are meant to represent the typical usage of home automation devices. They exemplify how some simple tasks and actions that people usually do can be automated, and how the automation can be adapted to everyday use.

To avoid variation, the same room setup will be used through all scenarios. There are some furniture, a door and a window with a blind. The only artificial source of light in all room, is a lamp on the ceiling. There is also a heater that has the power to heat the whole room. The Figure 43 represents an example of a room with this type of setup.

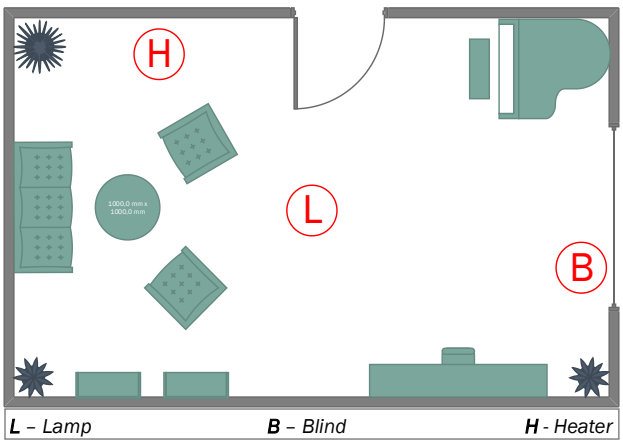


Figure 43 – A possible room setup to be used in the scenarios

In each of the subsequent scenarios, there are sensors that are aware of events and actuators that do specific actions. The objective is to demonstrate how these devices work in very specific situations, because they will be used later in more complex scenarios.

The following lists represents a catalogue of the sensors and actuators with their inputs and outputs:

Sensors' list

1. Light switch (with one button)
   - Input: Push/press
   - Output: Type of action (if pressed, the pressed time)
2. Movement sensor
   - Input: Movement
   - Output: Indication of movement
3. Daylight sensor
   - Input: Daylight
   - Output: Daylight intensity
4. Temperature sensor
   - Input: Temperature
   - Output: Temperature variation

Actuators' list

1. Light bulb
   - Input: Power amount
   - Output: Light intensity
2. Blind
   - Input: Action command
   - Output: Blind action
3. Heater
   - Input: Mode type
   - Output: Heat intensity

9.1.1.1 Click dimmer

Sensor: Light switch

Actuator: Light bulb

Description

The objective of this scenario is to provide more functions to the existing light bulb of the lamp. Instead of simply turning the light on and off, it would be useful to give the light switch the possibility to control the illuminance level. In order to do so, the following actions should be supported:

1. Turn the light on (max illuminance level);
2. Turn the light off;
3. Control the light intensity.

To fully support these actions, a special light switch should be used. This switch has a single push-button that supports push/press interaction and does all the previous actions:

1. Turn the light on
   - *Action required:* Push the light switch button
   - *Previous state:* Light off
   - *Subsequent state:* Light on
2. Turn the light off
   - *Action required:* Push the light switch button
   - *Previous state:* Light on

- *Subsequent state:* Light off

3. Control the light intensity
   - *Action required:* Press the light switch button
     This action depends on the previous state. If the illuminance level were raised, than the subsequent action will dim the light level. If the illuminance level were dimmed, than the subsequent action will raise the light level.

The light switch has a single push-button since this type of buttons are cheaper than rotary dimmer switches. The user can simply push the button to turn the light on or off, but if he wants to change the light intensity, the button must be pressed instead.

9.1.1.2 Occupancy sensor control

Sensor: Movement sensor

Actuator: Light bulb

Description

To save energy, the room should only have the light on when there are people inside. Furthermore, the light intensity should adapt itself to the user's presence, especially when the user is not in the room or stays there for a short period of time, thus reducing the amount of energy wasted.

This behavioral type is achieved through occupation sensors that directly control the light. This sensors should be placed high enough (e.g. at the room ceiling) to accurately detect the presence of people in the room.

If the sensor satisfies all the requirements, then it should perform the following actions:

1. When someone enters the rooms, the light is turned on at a pre-defined level (e.g. around 50%) and raises about 5% per minute, so the eyes of the user could adapt to the light;

2. When someone leaves the room, the light is not immediately turned off. Instead, it dims to a certain level (e.g. around 50%) and after a predetermined amount of time (e.g. 5 minutes) it turns off completely. This is useful when the user is leaving the room and there is not no light outside.

The previous actions only require one sensor, since the movement in the room is the only action that will be monitored.

9.1.1.3 Daylight harvesting

Sensor: Daylight sensor

Actuator: Light bulb

Description

When the outside light is not adequate to bright all room, the lamp should be used to compensate that. The goal of daylight harvesting is to sync the light illuminance level with the outside light, to provide the suitable amount of electric light to well-lit the room.

To detect the outside lux (luminous emittance), a light sensor should be placed in the room. This sensor can be located anywhere depending on the calibration of it, so the light bulb can be

turned on as soon as the outside lux is diminished. For this scenario, the blind is considered to be always open.

9.1.1.4 Blind control

Sensor: Daylight sensor

Actuator: Blind

Description

For this scenario, it will be used the blind, which is directly dependent of the outside illuminance level. It can have three states:

- Totally closed;
- Widely open;
- In sync with the light outside.

These are the three actions that can be used to control the blind and adapt the outside light to the user's purpose. The last action should filter the light in a way that prevents inside glare.

The light sensor should be the same used on the daylight harvesting scenario and could be located on the same place.

9.1.2 Concerns about the scenarios' evolution

The typical scenarios described above were used to demonstrate simple situations where home automation can be applied. Those scenarios can be combined with each other to provide more interesting situations, which will generate unexpected behaviors that must be resolved. This will allow to draw some conclusions about how the system should behave/act when facing a determined situation/conflict. These conclusions will be vital to understand how to manage priorities and conflicts; decide what should be done when there is a decision to make and identify the need to have other equipment involved, besides the sensors and actuators.

9.1.2.1 How important is the user?

To establish if the presence of the user has relevance, there will be used a scenario that combines the automatic behavior of the two sensors.

Sensors:

- Daylight sensor (S1)
- Movement sensor (S2)

Actuators:

- Light bulb (A1)

Description

For this scenario, the conditions are always the perfect ones. This means that the light outside is always enough to lit all the room. The behavior explained in the simple scenarios is pretty straight forward for each of the sensors:

- For S1, if there is light outside, A1 turns on and it turns off otherwise (Table 12).

Table 12 – How A1 behaves when there is light outside

	A1 state
Light outside (LO)	0
No light outside (!LO)	1

- For S2, it is the same thing. If there is presence in the room, A1 turns on and if there is not, it turns off (Table 13).

Table 13 – How A1 behaves when there is presence in the room

	A1 state
Presence (P)	1
No presence (!P)	0

Since the A1 is the same actuator for both sensors, the Table 12 and Table 13 can be combined into Table 14.

Table 14 – A1 behavior in response to S1 and S2

	LO	*!LO*
P	×	1
!P	0	×

The Table 14 shows that when there are two sensors involved, there are also conflict situations to solve. Before those, let's see the easiest ones:

- When there is no light outside (!LO) and there is presence in the room (P), the light bulb (A1) is on.
- When there is light outside (LO) and there is no presence in the room (!P), A1 is off.

Questions:

1. What should be done when there is light outside (LO) and there is presence in the room (P)?
2. And when there is not light outside (!LO) and there is no presence in the room (!P)?

Answers:

1. If there is light outside (LO) and the user is in the room, it does not make sense to have the light bulb (A1) on.
2. If there is not light outside (LO) and there is no user in the room (!P), then it does not make sense to have the light bulb (A1) on.

Conclusions

With the answers above, the Table 14 can be updated into the following result:

Table 15 – A1 custom behavior in response to S1 and S2

	LO	*!LO*
P	0	1
!P	0	0

From this explanation, there are some assumptions that can be made:

- The movement sensor (S2) has priority above the daylight sensor (S1), since when there is no presence, the light bulb is off.
- When the system detects a presence, the other sensor should be verified, in this case the daylight sensor (S1).

This example validates that the user presence has high-priority over the other sensors, which can also contribute to save some energy. The user presence does not solve all the overlapping situations, but has an important contribute to solve some of them.

9.1.2.2 Is there a need for a controller?

With the previous example, the light bulb (A1) could be only on or off, but what if the artificial light in the room could have different levels?

Sensors:

- Daylight sensor (S1)
- Movement sensor (S2)

Actuators:

- Light bulb (A1)

Description

This new scenario is an evolution of the previous one. In this case, the intensity of the light outside is variable, which means that A1 can be directly dependent of the S1. Considering that A1 can have five levels of intensity, the Table 16 summarizes an example of a possible behavior of A1.

Table 16 – A1 intensity level, depending on the outside light illuminance

Outside light intensity	*A1 light level*
[0-19]	100
[20-39]	75
[40-59]	50
[60-79]	25
[80-100]	0

Following the conclusions drawn in the Section 9.1.2.1, this behavior of A1 is only possible when there is presence in the room.

Questions:

1. How can the behavior of A1 be defined?
2. Who does the association between the sensors and actuators?

Answers:

1. The behavior of actuator A1 can be defined if there is an entity responsible for controlling it.
2. There is a need to have another element that makes the link between the sensor and the actuator, which means a controller that manages that connection will be required.

Conclusions

To support the described situation, the presence of controllers is essential:

- To bind together the sensor and the actuator, a controller is required.
- To analyze the data to/from a device, a controller can be necessary.

Figure 44 represents the interaction between a sensor and an actuator, with the use of controllers. As can be seen, the controller analyses the received information and after processing it, sends actions that controls the actuator.

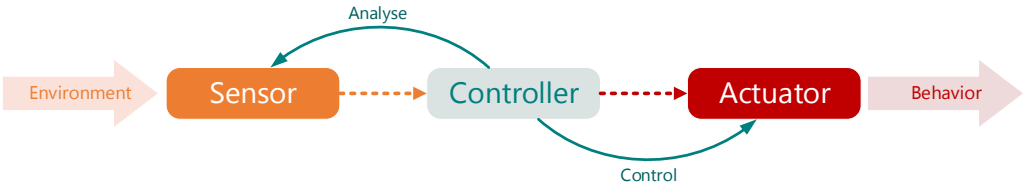


Figure 44 – Interaction between a sensor and an actuator

9.1.2.3 What is the impact of the user's actions?

When there is a system fully automated, there could be conflicts with the user's actions and the excessive automation, so it is vital to understand what is the role of the user in the system.

Sensors:

- Daylight sensor (S1)
- Light switch (S3)

Actuators:

- Light bulb (A1)

Description

This scenario has a conflict concerning the automated action provided by S1 and the manual action that can be executed using S3. The daylight sensor manages the light intensity of A1 depending on the intensity of the light outside, but the user can (at the same time) control the same actuator.

Questions:

1. Does it make sense to interrupt the system by manual actions?
2. If the system suffers that interruption, what will be its behavior?

Answers:

1. The user must always have the power to modify the system's comportment, so it makes sense that when a user manipulates a device directly, the system's automated behavior is interrupted.
2. When the system suffers an interruption, the automated behavior should stop. After that, there is another issue to solve, that is, how the system returns to its automated state. There are two possible ways:
    a. The system resumes its automated behavior, by user's order.
    b. When the system does not detect the user presence in the room, it assumes that after a certain time it can return to its automated behavior.

Conclusions

The user's actions have a great impact in the system's behavior, because they work like an interruption in the system's standard comportment. This means that the user's interruptions, and even his presence have priority over the system's behavior.

9.1.2.4 Is it possible to have scheduled actions?

For a HA system, the automated actions are an indispensable feature, but they depend on specific triggers to be executed (e.g. the presence of people in the room). A way to explore the potential of a system for this domain, is to include the possibility of having actions that can be scheduled. The problem is that the inclusion of scheduled actions may bring entropy to the system.

Sensors:

- Daylight sensor (S1)
- Movement sensor (S2)
- Temperature sensor (S4)

Actuators:

- Light bulb (A1)
- Blind (A2)

Description

In a hot summer day, it is usual for people to have the blind down to keep the house fresh. With the blind down, there is however, the necessity of turning on the light when the user is in the room. This behavior can be scheduled into the system to be executed at an interval of time (e.g. between 13h00 and 18h00).

Questions:

1. What happens if the user contradicts the scheduled behavior?
2. Could the system have different types of schedules?

Answers:

1. Like in the previous sections, the user's actions always have priority over the system's behavior, and the scheduling is no different.
2. There could be two different types of schedules:
    a. A permanent schedule that is created once and runs always at a specific time (e.g. at 23h00, dim all the lights in the house).
    b. A schedule that runs only once for a specific situation (e.g. today at 17h05, blink the lights twice).

Conclusions

With the inclusion of scheduling, it is clear that the user's actions always come first. The problem is which have more priority: user's scheduling or user's presence. Both of those features have more importance than the automated behavior of the system, but between them it is difficult to draw a conclusion.

9.1.2.5 Final conclusions

In the previous four sections, several deductions that helps to characterize the HA system were taken. The following list summarizes them:

- The presence sensor has great importance because it detects when a user is in the room, which means that some automated actions can have different behaviors.
- The presence of a controller to make the connection between the sensor(s) and the actuator(s) is essential, because there are some behaviors that should be programmed for particular situations. A controller for each device can also be required, if the behavior of that device is too specific (e.g. controlling the temperature of an air-conditioner).
- The user's actions always have priority above the system's behavior.

The scheme in Figure 45 shows a possible way to rank the priority of events.

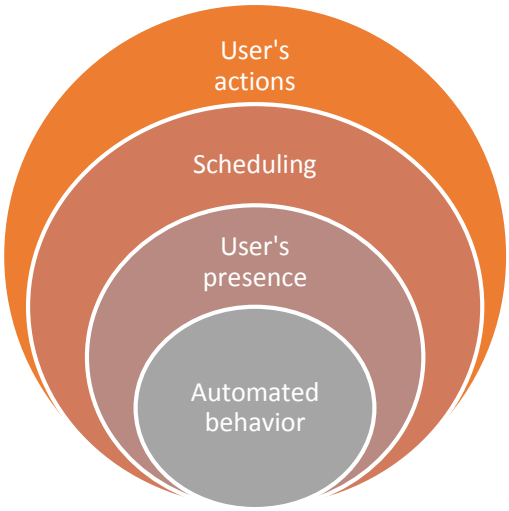


Figure 45 – Hierarchy of events in the system. More priority represented by the outside circle

To explain the order of events in Figure 45, it is relevant to review each one of them:

1. **User's actions –** The actions performed by the user are more important than the others, since the user has the ultimate control over the system.
2. **Scheduling –** The doubt in this ranking was between the scheduling and the user's presence. If a user schedules something, that planned event should occur even if the user is present. If he wants to interrupt the scheduled event, he can do so by executing an action.
3. **User's presence –** The detection of the user presence is important, because it can avoid some situations where the automated behavior does not make sense.
4. **Automated behavior –** This is the normal interaction between the sensors and the actuators when the previous events are not being executed.

9.2 HEATING ROOM CLASS DIAGRAM

This section contemplates the class diagram defined for the Heating room scenario.

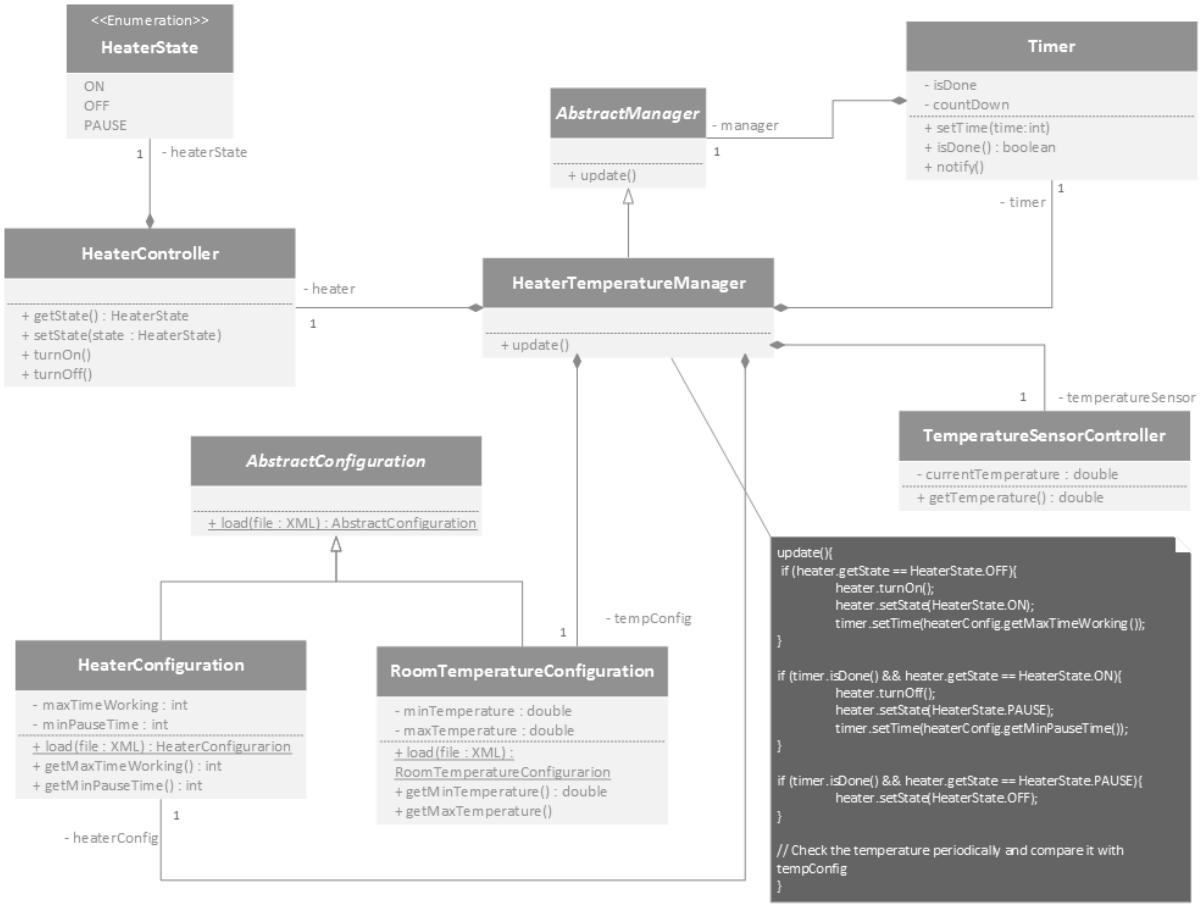


Figure 46 – Class diagram of the Heating room scenario

9.3 LABVIEW DASHBOARD

Overview of the dashboard used in LabVIEW platform.

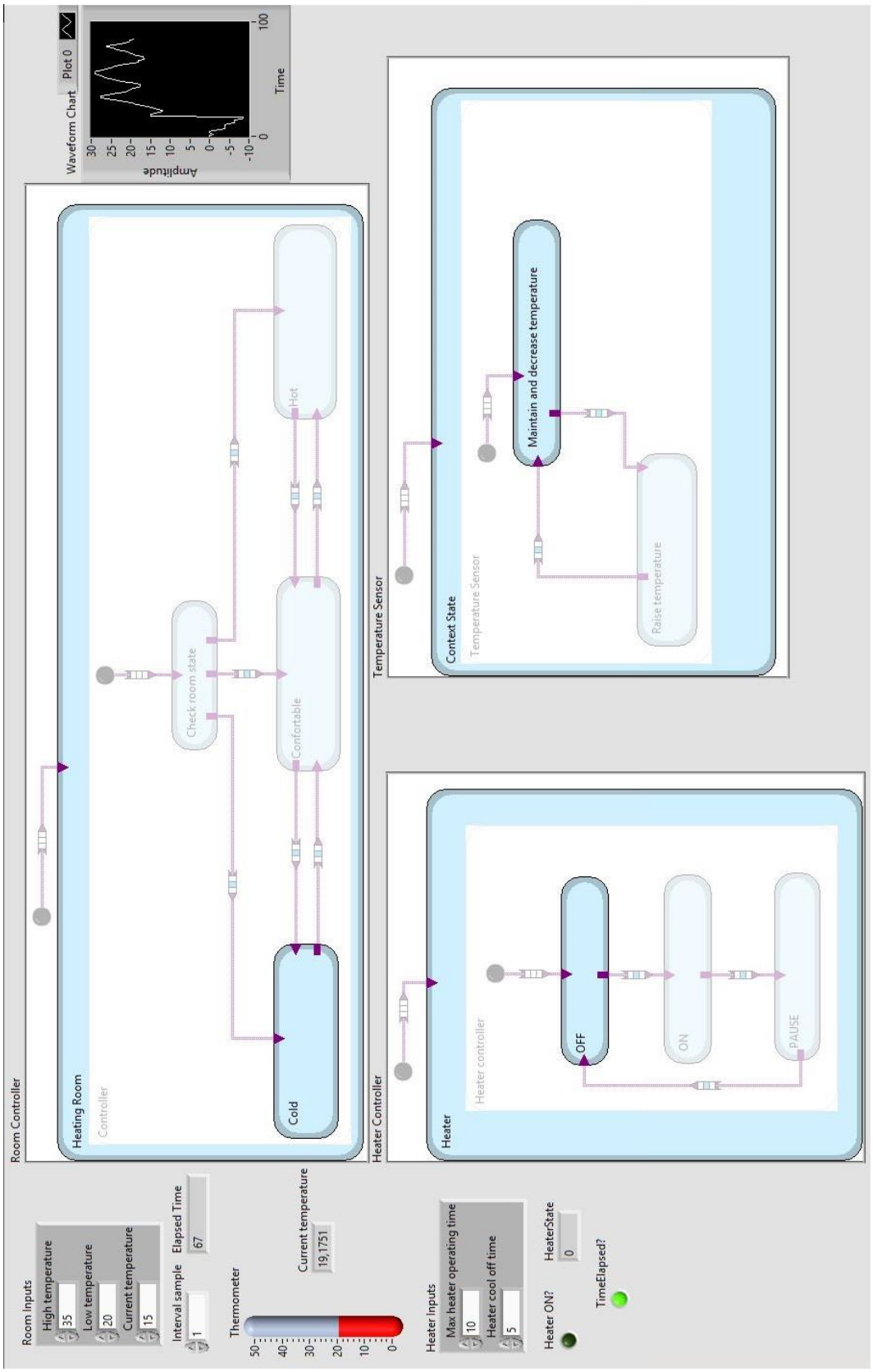


Figure 47 – Dashboard of the implementation in LabVIEW

9.4 MODEL DEFINITION

The following sub-section represent the Feature and Domain models.

9.4.1 Feature Model

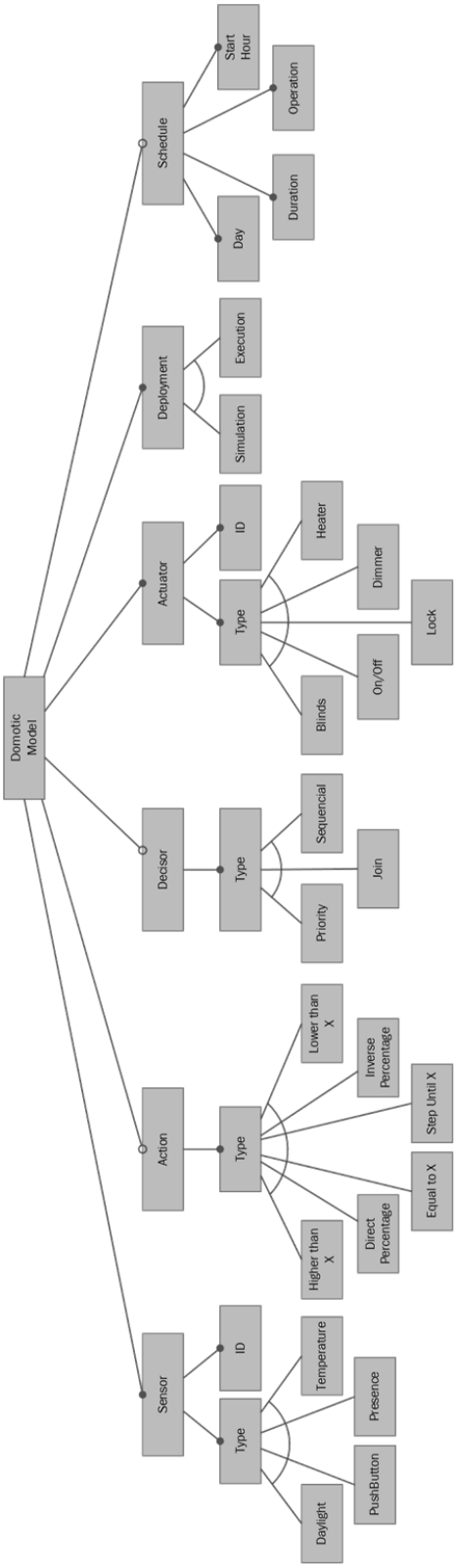


Figure 48 – Feature Model

9.4.2 Domain Model

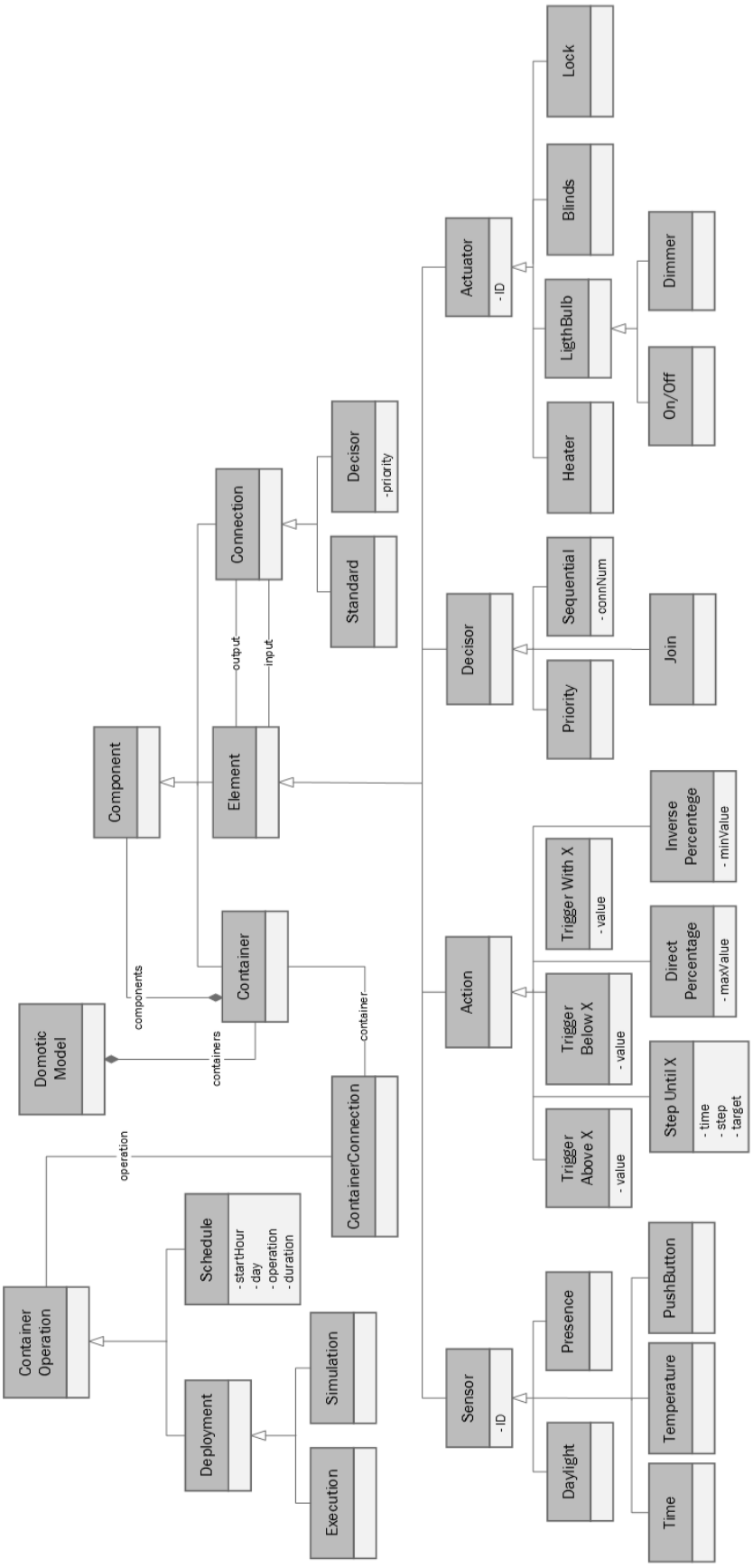


Figure 49 – Domain Model

9.5 ECORE DIAGRAM

This section contemplates the Ecore diagram used to build the DSL.

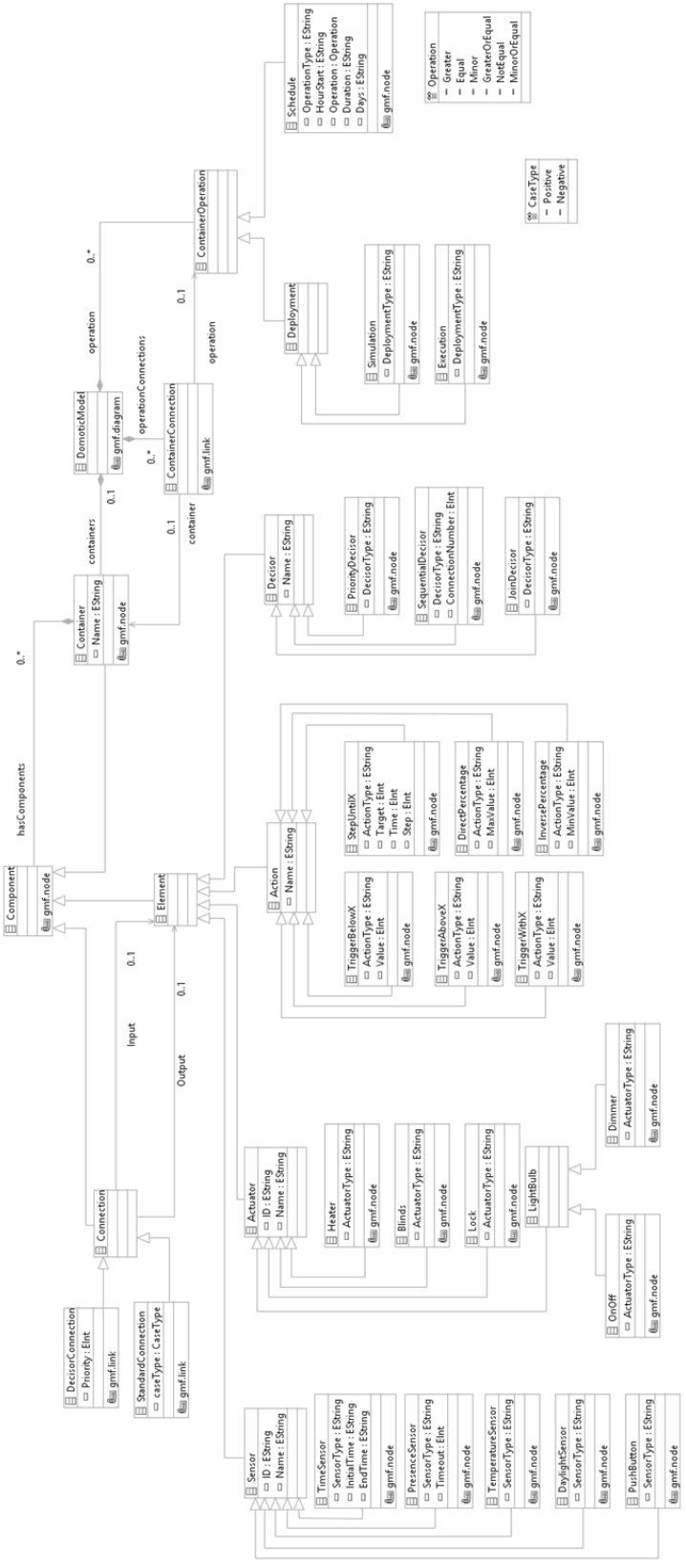


Figure 50 – Ecore diagram of the DSL

9.6 EMF

The EMF file is used to describe by text the metamodel concepts, rules and properties.

Listing 7 – Description of the metamodel concepts in the EMF file

```
@namespace(uri="Domotic_3", prefix="Domotic_3")
package Domotic_3;

@gmf.diagram
class DomoticModel {
  val Container containers;
  val ContainerOperation [*] operation;
  val ContainerConnection [*] operationConnections;
}

@gmf.node
abstract class Component {
}

@gmf.node(label="Name", label.pattern="{0}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Container_24.gif")
class Container extends Component {
  attr String Name;

  @gmf.compartment
  val Component[*] hasComponents;
}

abstract class Element extends Component {
}

abstract class Sensor extends Element {
  attr String ID;
  attr String Name;
}

abstract class Actuator extends Element {
  attr String ID;
  attr String Name;
}

abstract class Action extends Element {
  attr String Name = "Action";
}

abstract class Decisor extends Element {
  attr String Name = "Decisor";
}

abstract class Connection extends Component {
  ref Element Input;
  ref Element Output;
}

@gmf.link(label="caseType", source="Input", target="Output", style="solid", width="2",
target.decoration="arrow", color="0,0,0", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/GlobalConnection.gif")
class StandardConnection extends Connection {
  attr CaseType caseType;
}
```

```
enum CaseType{
       Positive = 1;
       Negative = 0;
}

@gmf.link(label="Priority", source="Input", target="Output", style="dot", width="2",
target.decoration="arrow", label.icon="true", tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/ConnectionToDecisors.gif")
class DecisorConnection extends Connection {
  attr int Priority;
}

@gmf.node(label="Name,SensorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/TimeSensor_24.gif")
class TimeSensor extends Sensor {
  readonly attr String SensorType = "Time Sensor";
  attr String InitialTime = "00:00";
  attr String EndTime = "00:00";
}

@gmf.node(label="Name,SensorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Presence_Sensor_24.gif")
class PresenceSensor extends Sensor {
  readonly attr String SensorType = "Presence Sensor";
  attr int Timeout = 5;
}

@gmf.node(label="Name,SensorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Temperature_Sensor_24.gif")
class TemperatureSensor extends Sensor {
  readonly attr String SensorType = "Temperature Sensor";
}

@gmf.node(label="Name,SensorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Daylight_Sensor_24.gif")
class DaylightSensor extends Sensor {
  readonly attr String SensorType = "Daylight Sensor";
}

@gmf.node(label="Name,SensorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Push_Button_Sensor_24.gif")
class PushButton extends Sensor {
  readonly attr String SensorType = "Push Button Sensor";
}

@gmf.node(label="Name,ActuatorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Heater_24.gif")
class Heater extends Actuator {
  readonly attr String ActuatorType = "Heater";
}

abstract class LightBulb extends Actuator {
}

@gmf.node(label="Name,ActuatorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/On_Off_24.gif")
class OnOff extends LightBulb {
  readonly attr String ActuatorType = "On/Off Light";
}
```

```
@gmf.node(label="Name,ActuatorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Dimmer_24.gif")
class Dimmer extends LightBulb {
  readonly attr String ActuatorType = "Dimmer Light";
}

@gmf.node(label="Name,ActuatorType", label.pattern="{0} : {1}", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Blinds_24.gif")
class Blinds extends Actuator {
  readonly attr String ActuatorType = "Blinds";
}

@gmf.node(label="ActuatorType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Lock_24.gif")
class Lock extends Actuator {
  readonly attr String ActuatorType = "Lock";
}

@gmf.node(label="ActionType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/TriggerBelowX_24.gif")
class TriggerBelowX extends Action {
  readonly attr String ActionType = "Trigger Below X";
  attr int Value;
}

@gmf.node(label="ActionType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/TriggerAboveX_24.gif")
class TriggerAboveX extends Action {
  readonly attr String ActionType = "Trigger Above X";
  attr int Value;
}

@gmf.node(label="ActionType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/TriggerWithX_24.gif")
class TriggerWithX extends Action {
  readonly attr String ActionType = "Trigger With X";
  attr int Value;
}

@gmf.node(label="ActionType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/StepUntilX_24.gif")
class StepUntilX extends Action {
  readonly attr String ActionType = "Step Until X";
  attr int Target;
  attr int Time;
  attr int Step;
}

@gmf.node(label="ActionType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/DirectPercentage_24.gif")
class DirectPercentage extends Action {
  readonly attr String ActionType = "Direct Percentage";
  attr int MaxValue;
}
```

```
@gmf.node(label="ActionType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/InversePercentage_24.gif")
class InversePercentage extends Action {
  readonly attr String ActionType = "Inverse Percentage";
  attr int MinValue;
}

@gmf.node(label="DecisorType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/PriorityDecisor_24.gif")
class PriorityDecisor extends Decisor {
  readonly attr String DecisorType = "Priority Decisor";
}

@gmf.node(label="DecisorType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/SequentialDecisor_24.gif")
class SequentialDecisor extends Decisor {
  readonly attr String DecisorType = "Sequential Decisor";
  attr int ConnectionNumber = 1;
}

@gmf.node(label="DecisorType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/JoinDecisor_24.gif")
class JoinDecisor extends Decisor {
  readonly attr String DecisorType = "Join Decisor";
}

@gmf.link(source="container", target="operation", style="solid", width="2",
target.decoration="arrow", color="10,150,0", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/ContainerConnection.gif")
class ContainerConnection {
  ref Container container;
  ref ContainerOperation operation;
}

abstract class ContainerOperation {
}

abstract class Deployment extends ContainerOperation {
}

@gmf.node(label="DeploymentType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Simulation_24.gif")
class Simulation extends Deployment {
  readonly attr String DeploymentType = "Simulation";
}

@gmf.node(label="DeploymentType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Execution_24.gif")
class Execution extends Deployment {
  readonly attr String DeploymentType = "Execution";
}
```

```
@gmf.node(label="OperationType", label.icon="true",
tool.small.bundle="Thesis_Sample_v3.edit",
tool.small.path="/icons/full/obj16/Schedule_24.gif")
class Schedule extends ContainerOperation{
	readonly attr String OperationType = "Schedule";
	attr String HourStart = "00:00";
	attr Operation Operation;
	attr String Duration;
	attr String Days = "MON,TUE,WED,THU,FRI,SAT,SUN";
}

enum Operation{
	Greater = 0;
	Equal = 1;
	Minor = 2;
	GreaterOrEqual = 3;
	NotEqual = 4;
	MinorOrEqual = 5;
}
```

9.7 EVL

The EVL file contains all the well-formedness rules that are used to check if the behavior is correctly implemented.

Listing 8 – The well-formedness rules that constitute the EVL file

```
context Container {

constraint hasName {
	check : self.Name.isDefined()
	message : 'You must define a name for this Container'
	fix {
		title : 'Forcing a name'
		do {
			self.Name := 'Some_Container';
		}
	}
}

constraint hasACorrectName {
	guard: self.satisfies('hasName')
	check : self.Name.matches("^[A-Za-z0-9_]+$")
	message : 'The name of the Container must be a single word, with only letters
and/or numbers'
}

constraint checkContainer {
	check : not self.hasComponents.exists(t|t.isKindOf(Container))
	message : 'A Container cannot have a container inside'
	fix {
		title : 'Removing Container'
		do {
			for(p in self.hasComponents.select(t|t.isKindOf(Container))){
				delete p;
			}
		}
	}
}

constraint hasAllTheNecessaryComponents {
	check : (self.hasComponents.exists(t|t.isKindOf(Sensor)) and
self.hasComponents.exists(t|t.isKindOf(Actuator)))
	message : 'A Container must have at least one Sensor and one Actuator'
}

constraint doesNotSupportIsolatedBehavior {
	check : (Element.allInstances.size()-1 <= Connection.allInstances.size())
	message : 'A Container cannot have an isolated behavior. Consider the usage of a
Decisor or the separation in two Containers'
}

constraint shouldHaveAtLeastADeploymentType {
	check:
not(ContainerOperation.allInstances.select(c|c.isKindOf(Deployment)).size() == 0)
	message : 'A Container must have a Deployment type'
}

constraint canOnlyHaveADeploymentType {
	check:
not(ContainerOperation.allInstances.select(c|c.isKindOf(Deployment)).size() > 1)
	message : 'A Container only supports a Deployment type'
}
```

```
constraint canOnlyHaveASchedule {
	check: not(ContainerOperation.allInstances.select(c|c.isKindOf(Schedule)).size()
> 1)
	message : 'A Container only supports a Schedule'
}
}

context Element {
constraint isNotIsolated {
	check : (Connection.allInstances.exists(i|i.Input = self) or
Connection.allInstances.exists(i|i.Output = self))
	message : 'An Element must have a connection with somenone'
}
}

context Sensor {

constraint hasName {
	check : self.Name.isDefined()
	message : 'You must define a name for this Sensor'
	fix {

		title : 'Forcing a name'

		do {
			self.Name := 'Some_Sensor';
		}
	}
}

constraint hasACorrectName {
	guard: self.satisfies('hasName')
	check : self.Name.matches("^[A-Za-z0-9_]+$")
	message : 'The name of the Sensor must be a single word, with letters, numbers
and underscore'
}

constraint shouldNotHaveTheSameName {
	guard: Sensor.allInstances.select(s|s.Name.isDefined()).size() ==
Sensor.allInstances.size()
	check : not (Sensor.allInstances.select(s|s.Name.equals(self.Name)).size()>1)
	message : 'The name of each Sensor must be different'
}
}

context Actuator {

constraint hasName {
	check : self.Name.isDefined()
	message : 'You must define a name for this Actuator'
	fix {
		title : 'Forcing a name'
		do {
			self.Name := 'Some_Actuator';
		}
	}
}

constraint hasACorrectName {
	guard: self.satisfies('hasName')
	check : self.Name.matches("^[A-Za-z0-9_]+$")
	message : 'The name of the Actuator must be a single word,  with letters,
numbers and underscore'
}
```

```
constraint shouldNotHaveTheSameName {
	guard: Actuator.allInstances.select(s|s.Name.isDefined()).size() ==
Actuator.allInstances.size()
	check : not (Actuator.allInstances.select(s|s.Name.equals(self.Name)).size()>1)
	message : 'The name of each Actuator must be different'
}

constraint shouldOnlyHaveALineConnected {
	guard: Actuator.allInstances.select(s|s.Name.isDefined()).size() ==
Actuator.allInstances.size()
	check :
not(StandardConnection.allInstances.select(i|i.Output.isKindOf(Actuator)).select(k|k.Out
put.Name.equals(self.Name)).size()>1)
	message : 'An Actuator only supports one behavior. If you are trying to connect
two, consider the usage of a Decisor'
}
}

context Decisor{
constraint canOnlyHaveOneOutput {
	check : not (StandardConnection.all.select(d|d.Input == self).size() > 1)
	message : 'A Decisor can only have one output connection'
}

constraint canOnlyConnectToAnActuator {
	check : not
(StandardConnection.all.select(d|d.Input.isKindOf(Decisor)).select(a|a.Output.isKindOf(A
ctuator)) > 1)
	message : 'A Decisor can only connect to an Actuator'
}

constraint canOnlyHaveDecisorConnectionsToHimself {
	check : not (StandardConnection.all.exists(d|d.Output.isKindOf(Decisor)))
	message : 'Only a Connection to Decisor can be connected to a Decisor'
}
}

context PriorityDecisor{
constraint canOnlyHaveTwoInputs {
	check : not (DecisorConnection.all.select(d|d.Output == self).size() > 2)
	message : 'A Priority Decisor can only have two input connections'
}
}

context SequentialDecisor{
constraint connectionNumberIsValid {
	check : not (DecisorConnection.all.select(d|d.Output == self).size() <
self.ConnectionNumber)
	message : 'The property ConnectionNumber must be valid'
}
}

context StandardConnection{
constraint theOutputCannotBeADecisor {
	check : not (self.Output.isKindOf(Decisor))
	message : 'A Global Connection cannot have a Decisor as output. Change it for a
Connection to Decisor'
}

constraint sensorCompatibilityWithNegativeConnection{
	guard: self.Input.isKindOf(Sensor) and self.caseType.value == 0
	check : (self.sensorCompatibilityWithNegativeConnection())
	message : 'This type of Sensor does not support Negative behavior'
}
```

```
//Compatibilities
constraint sensorCompatibility{
	guard: self.Input.isKindOf(Sensor) and self.Output.isKindOf(Action)
	check : (self.sensorCompatibility())
	message : 'A Sensor '+ self.Input.SensorType + ' is not compatible with an
Action ' + self.Output.ActionType
}

constraint actionCompatibility{
	guard: self.Input.isKindOf(Action) and self.Output.isKindOf(Action)
	check : (self.actionCompatibility())
	message : 'An action '+ self.Input.ActionType + ' is not compatible with an
Action ' + self.Output.ActionType
}

constraint actuatorCompatibility{
	guard: self.Output.isKindOf(Actuator) and not self.Input.isKindOf(Decisor)
	check : (self.actuatorCompatibility())
	message : 'This element is not compatible with an Actuator ' +
self.Output.ActuatorType
}
}

context DecisorConnection{
constraint shouldFollowACertainOrder {
	check : self.checkOrderOfConnection()
	message : 'A Connection to a Decisor must follow a specific order, beginning at
1, with increments of 1 unit'
}

constraint theOutputShouldBeADecisor {
	check : not (self.Output.isKindOf(Sensor) or self.Output.isKindOf(Actuator) or
self.Output.isKindOf(Action))
	message : 'A Connection to Decisor should have a Decisor as output. Change it
for a Global Connection'
}
}

operation DecisorConnection checkOrderOfConnection() : Boolean {
var totalConnections = DecisorConnection.all.select(d|d.Output == self.Output);
var numbersSequence : OrderedSet;

for(decisor in totalConnections){
	numbersSequence.add(decisor.Priority);
}

if(not(numbersSequence.size() == totalConnections.size())){
	return false;
}

var idxAux = 1;

while(idxAux <= totalConnections.size()){
	if(not (numbersSequence.includes(idxAux))){
		return false;
	}
	idxAux = idxAux + 1;
}

return true;
}
```

```
operation StandardConnection sensorCompatibilityWithNegativeConnection() : Boolean {
switch (self.Input.type().Name){
	case "PresenceSensor":
		return true;
	case "TemperatureSensor":
		return false;
	case "DaylightSensor":
		return false;
	case "PushButton":
		return true;
	case "TimeSensor":
		return false;
}
}

operation StandardConnection sensorCompatibility() : Boolean {
switch (self.Input.type().Name){
	case "PresenceSensor":
		return (self.Output.isKindOf(TriggerWithX));
	case "TemperatureSensor":
		return (not(self.Output.isKindOf(TriggerWithX) or
self.Output.isKindOf(StepUntilX)));
	case "DaylightSensor":
		return (not(self.Output.isKindOf(TriggerWithX) or
self.Output.isKindOf(StepUntilX)));
	case "PushButton":
		return (not( self.Output.isKindOf(StepUntilX)));
	case "TimeSensor":
		return (self.Output.isKindOf(TriggerWithX));
}
}

operation StandardConnection actionCompatibility() : Boolean {
switch (self.Input.type().Name){
	case "TriggerWithX":
		return (not(self.Output.isKindOf(TriggerBelowX) or
				self.Output.isKindOf(TriggerAboveX) or
				self.Output.isKindOf(TriggerWithX)));
	case "StepUntilX":
		return (not(self.Output.isKindOf(TriggerWithX) or
				self.Output.isKindOf(StepUntilX)));
	case "TriggerBelowX":
		return (self.Output.isKindOf(TriggerWithX));
	case "TriggerAboveX":
		return (self.Output.isKindOf(TriggerWithX));
	case "DirectPercentage":
		return not(self.Output.isKindOf(TriggerWithX) or
				self.Output.isKindOf(StepUntilX) or
				self.Output.isKindOf(DirectPercentage) or
				self.Output.isKindOf(InversePercentage));
	case "InversePercentage":
		return not(self.Output.isKindOf(TriggerWithX) or
				self.Output.isKindOf(StepUntilX) or
				self.Output.isKindOf(DirectPercentage) or
				self.Output.isKindOf(InversePercentage));

	}
}
```

```
operation StandardConnection actuatorCompatibility() : Boolean {
switch (self.Output.type().Name){
	case "OnOff":
		return ((self.Input.isKindOf(Action) and
((self.Input.isKindOf(TriggerBelowX) or

	self.Input.isKindOf(TriggerAboveX))))

	or
				(self.Input.isKindOf(Sensor) and
((self.Input.isKindOf(PresenceSensor)))));
	case "Dimmer":
		return ((self.Input.isKindOf(Action) and
not((self.Input.isKindOf(TriggerBelowX) or

	self.Input.isKindOf(TriggerAboveX))))

	or
				(self.Input.isKindOf(Sensor) and
not((self.Input.isKindOf(PresenceSensor)))));
	case "Heater":
		return ((self.Input.isKindOf(Action) and
((self.Input.isKindOf(TriggerBelowX) or

	self.Input.isKindOf(TriggerAboveX))))

	or
				(self.Input.isKindOf(Sensor) and
((self.Input.isKindOf(PresenceSensor)))));
	case "Blinds":
		return ((self.Input.isKindOf(Action) and
not((self.Input.isKindOf(TriggerBelowX) or

	self.Input.isKindOf(TriggerAboveX))))

	or
				(self.Input.isKindOf(Sensor) and
not((self.Input.isKindOf(PresenceSensor)))));
	case "Lock":
		return ((self.Input.isKindOf(Action) and
((self.Input.isKindOf(TriggerBelowX) or

	self.Input.isKindOf(TriggerAboveX))))

	or
				(self.Input.isKindOf(Sensor) and
((self.Input.isKindOf(PresenceSensor)))));
}

}
```

9.8 EGL

The EGL file holds the code that generates the text to be used in both simulation and execution platforms. Since the code is too extensive, the next two sub-section only have an example of the code used for simulation and for execution.

9.8.1 Simulation code

Listing 9 – Simulink_templates.egl

```
	[%
	operation initBlockDiagram(systemName : String){%]
%Init block diagram
sys = '[%=systemName%]';
new_system(sys);
open_system(sys);
root = sfroot;
blockDiagram = root.find('-isa','Simulink.BlockDiagram');

%Real Time for simulation
add_block('utility/Soft Real Time', [sys '/Real Time']);
set_param([sys '/Real Time'],'Position', [30 20 30+90 20+30]);
set_param([sys '/Real Time'],'x', '1');

%Real Time Sync
add_block('rtwinlib/Real-Time Synchronization', [sys '/Real Time Sync']);
set_param([sys '/Real Time Sync'],'Position', [150 20 150+90 20+30]);
	[% } %]

	[%
	operation initChart(chartName:String, chartNumber:Integer){%]
% Chart init
add_block('sflib/Chart', [sys '/[%=chartName%]']);
chart[%=chartNumber%] = blockDiagram.find('-isa','Stateflow.Chart', '-and', 'Name',
'[%=chartName%]');
	[% } %]

	[%
	operation createVariables(inputs:Collection, outputs:Collection,
internals:Collection, chartNumber:Integer) { %]
		[%
		var i : Integer = 0;

		for (input in inputs) { %]
input[%=i%] = Stateflow.Data(chart[%=chartNumber%]);
input[%=i%].Scope = 'INPUT_DATA';
input[%=i%].Name = '[%=input%]';
			[%i=i+1;%]
		[% } %]

		[%i=0;%]
		[%
		for (output in outputs) { %]
output[%=i%] = Stateflow.Data(chart[%=chartNumber%]);
output[%=i%].Scope = 'OUTPUT_DATA';
output[%=i%].Name = '[%=output%]';
			[%i=i+1;%]
		[% } %]
```

```
[%i=0;%]

               [%
               for (internal in internals) { %]
internal[%=i%] = Stateflow.Data(chart[%=chartNumber%]);
internal[%=i%].Scope = 'LOCAL_DATA';
internal[%=i%].Name = '[%=internal%]';
                       [%i=i+1;%]
               [% } %]
       [% } %]

       [%
       operation statesCreation(states:Collection, statesConnections:Collection,
statesInnerInstructions:Collection, statesPosition:Collection, chartNumber:Integer)
{ %]
% States creation
height = 60;
width = 200;
initPosX = 150;
initPosY = 50;
offsetX = 150+width;
offsetY = 150;
spaceBetweenStates = 150;
centerLabel = (spaceBetweenStates - height)/2;
               [%
               var i : Integer = 0;
               var statesPositionFinal = defineStatesPositions(states,
statesConnections, statesPosition);

               for (state in states) { %]
       %[%=state%]
       state[%=state%]ID = [%=i%];
       state[%=state%] = Stateflow.State(chart[%=chartNumber%]);
       state[%=state%].Name = '[%=state%]';
       state[%=state%].Position = [%=statesPositionFinal.at(i)%];
       state[%=state%].LabelString =
sprintf('[%=state%]\n[%=statesInnerInstructions.at(i)%]');

                       [%i=i+1;%]
               [% } %]
       [% } %]

       [%
       operation transitionsCreation(states:Collection,
statesConnections:Collection, statesConnectionsPosition:Collection,
statesConnectionsLabels:Collection, chartNumber:Integer) { %]
% Transitions creation
               [%
               var i : Integer = 0;
               var j : Integer = 0;
               var k : Integer = 0;

               for (state in states) {
                       for (connection in statesConnections.at(i)) { %]
                       [% var stateToConnectName = states.at(connection);%]
       %[%=state%] -> [%=stateToConnectName%]
       trans[%=state%]_[%=stateToConnectName%] =
Stateflow.Transition(chart[%=chartNumber%]);
       trans[%=state%]_[%=stateToConnectName%].Source = state[%=state%];
```

```
                              [%if(state.equals(stateToConnectName)){%]
       junction = Stateflow.Junction(chart[%=chartNumber%]);
       trans[%=state%]_[%=stateToConnectName%].Destination = junction;
       junctionTrans = Stateflow.Transition(chart[%=chartNumber%]);
       junctionTrans.Source = junction;
       junctionTrans.Destination = state[%=stateToConnectName%];
       junctionTrans.DestinationOClock =
[%=statesConnectionsPosition.at(i).at(j).second()%];
                              [%}else{%]
       trans[%=state%]_[%=stateToConnectName%].Destination =
state[%=stateToConnectName%];
       trans[%=state%]_[%=stateToConnectName%].DestinationOClock =
[%=statesConnectionsPosition.at(i).at(j).second()%];
                              [%}%]

       trans[%=state%]_[%=stateToConnectName%].LabelString =
'[%=statesConnectionsLabels.at(i).at(j)%]';
       trans[%=state%]_[%=stateToConnectName%].SourceOClock =
[%=statesConnectionsPosition.at(i).at(j).first()%];
       pos = trans[%=state%]_[%=stateToConnectName%].LabelPosition;
       labelLen = length(trans[%=state%]_[%=stateToConnectName%].LabelString);
       pos(1) = trans[%=state%]_[%=stateToConnectName%].MidPoint(1) - (3*labelLen);
       pos(2) = trans[%=state%]_[%=stateToConnectName%].MidPoint(2) - 15;
       trans[%=state%]_[%=stateToConnectName%].LabelPosition = [pos(1) pos(2) 0 0];

                              [%if(state.equals(stateToConnectName)){%]
       auxPos = [trans[%=state%]_[%=stateToConnectName%].SourceEndPoint(1)
trans[%=state%]_[%=stateToConnectName%].SourceEndPoint(2);junctionTrans.DestinationEn
dPoint(1) junctionTrans.DestinationEndPoint(2)];
       d = pdist(auxPos,'euclidean');
       midPoint = [(trans[%=state%]_[%=stateToConnectName%].SourceEndPoint(1) +
junctionTrans.DestinationEndPoint(1))/2
(trans[%=state%]_[%=stateToConnectName%].SourceEndPoint(2) +
junctionTrans.DestinationEndPoint(2))/2];
       jPos1 = midPoint(1) + d;
       jPos2 = midPoint(2) + d;
       junction.Position.Center = [jPos1 jPos2];
       trans[%=state%]_[%=stateToConnectName%].SourceOClock =
[%=statesConnectionsPosition.at(i).at(j).first()%];
       junctionTrans.DestinationOClock =
[%=statesConnectionsPosition.at(i).at(j).second()%];
                              [%}%]
                     [%j=j+1;%]
                     [% } %]

                     [%i=i+1;%]
                     [%j=0;%]
              [% } %]
       [% } %]

       [%
       operation addDefaultState(firstStateName:String, chartNumber:Integer) { %]
% Add a default transition to state[%=firstStateName%]
defTrasns[%=firstStateName%] = Stateflow.Transition(chart[%=chartNumber%]);
defTrasns[%=firstStateName%].Destination = state[%=firstStateName%];
defTrasns[%=firstStateName%].DestinationOClock = 0;
xsource = state[%=firstStateName%].Position(1)+width/2;
ysource = state[%=firstStateName%].Position(2)-height/2;
defTrasns[%=firstStateName%].SourceEndPoint = [xsource ysource];
defTrasns[%=firstStateName%].MidPoint = [xsource ysource+15];
       [% } %]
```

```
	[%
operation defineStatesPositions(states:Collection, statesConnections:Collection,
statesPosition:Collection) : Collection { %]
	[%	var statesPositionsFinal : Sequence;%]
	[%
		var i : Integer = 0;
		var j : Integer = 0;
		for (state in states) { %]
			[%statesPositionsFinal.add("[(initPosX+offsetX*" +
statesPosition.at(i).first() + ") (initPosY+offsetY*" + statesPosition.at(i).second()
+ ") width height]");%]
			[%i=i+1;%]
		[% } %]
	[% return statesPositionsFinal;%]
[% } %]
```

Listing 10 – Simulink_Actuator_Generation.egl

```
[% //Dimmer Light Bulb
operation dimmerLightBulb(actuatorName:String, posX:Integer, posY:Integer) { %]
%Output block creation
add_block('gauges_gmslib/Percent Indicators/Generic Percent',[sys
'/[%=actuatorName%]']);
set_param([sys '/[%=actuatorName%]'],'Position', [[%=posX%] [%=posY%] [%=posX%]+70
[%=posY%]+70]);
[% } %]

[% //On_Off Light Bulb
operation onOffLightBulb(actuatorName:String, posX:Integer, posY:Integer)  { %]
%Output block creation
add_block('gauges_gmslib/On Off Gauges/Light Bulb',[sys '/[%=actuatorName%]']);
set_param([sys '/[%=actuatorName%]'],'Position', [[%=posX%] [%=posY%] [%=posX%]+70
[%=posY%]+70]);
[% } %]

[% //Heater State
operation heaterState(actuatorName:String, posX:Integer, posY:Integer) { %]
%Output block creation
add_block('gauges_gmslib/LEDs/Generic LED',[sys '/[%=actuatorName%]']);
set_param([sys '/[%=actuatorName%]'],'Position', [[%=posX%] [%=posY%] [%=posX%]+70
[%=posY%]+70]);
[% } %]

[% //Blinds
operation blinds(actuatorName:String, posX:Integer, posY:Integer) { %]
%Output block creation
add_block('gauges_gmslib/Linear Gauges/Generic Linear Gauge',[sys
'/[%=actuatorName%]']);
set_param([sys '/[%=actuatorName%]'],'Position', [[%=posX%] [%=posY%] [%=posX%]+70
[%=posY%]+70]);
[% } %]

[% //Lock
operation lock(actuatorName:String, posX:Integer, posY:Integer) { %]
%Output block creation
add_block('gauges_gmslib/On Off Gauges/Lock',[sys '/[%=actuatorName%]']);
set_param([sys '/[%=actuatorName%]'],'Position', [[%=posX%] [%=posY%] [%=posX%]+70
[%=posY%]+70]);
[% } %]
```

Listing 11 – Script_Generation.egl

```
[%
import "Simulink_templates.egl";
import "Simulink_Action_Generation.egl";
import "Simulink_Sensor_Generation.egl";
import "Simulink_Decisor_Generation.egl";
import "Simulink_Actuator_Generation.egl";
%]

[% operation generateScript(){%]
[%
var actionsChartMap : Map;
var actionName : String = "";
var decisorName : String = "";
var constantStrings : Sequence;
var actionNumber = 0;
var decisorNumber = 0;
var countDecisorInputs = 0;

var posXSensors = 30;
var posXActions = 500;
var posXDecisors = 700;
var posXActuators = 900;

var posYSensors = 70;
var posYActions = 70;
var posYDecisors = 70;
var posYActuators = 70;

var width = 150;
var height = 100;
%]

[% var container = Container.allInstances().first();%]

[%=initBlockDiagram(container.Name)%]
[% var sensorIdx := 0;%]

[% for (sensor in container.hasComponents.select(i|i.isKindOf(Sensor))) {%]
	[%switch (sensor.SensorType){%]
		[%case "Presence Sensor":%]
			[%=presenceSensorSimulation(sensor.Name, posXSensors,
posYSensors, sensor.Timeout)%]
		[%case "Temperature Sensor":%]
			[%=temperatureSensorSimulation(sensor.Name, posXSensors,
posYSensors)%]
		[%case "Daylight Sensor":%]
			[%=daylightSensorSimulation(sensor.Name, posXSensors,
posYSensors)%]
		[%case "Push Button Sensor":%]
			[%=pushButtonSimulation(sensor.Name, posXSensors,
posYSensors)%]
		[%case "Time Sensor":%]
			[%=timeSensorSimulation(sensor.Name, posXSensors,
posYSensors)%]
	[%}%]

	[%posYSensors = posYSensors + height + 30;%]
[% } %]
```

```
[% for (action in container.hasComponents.select(i|i.isKindOf(Action))) { %]
	[%if(action.Name == null){
		action.Name = "Action" + actionNumber;
	}
	else{
		action.Name = action.Name + actionNumber;
	}%]

	[%switch (action.ActionType){%]
		[%case "Trigger With X":%]
			[%actionName = equalsValue(action.Name, actionNumber);%]
			[%constantStrings = createConstantAction(action.Value,
actionName, actionNumber, posXActions, posYActions);%]
		[%case "Step Until X":%]
			[%actionName = stepUntilValue(action.Name, actionNumber);%]
			[%constantStrings =
createStepUntilValueConstants(action.Target, action.Time, action.Step, actionName,
actionNumber, posXActions, posYActions);%]
		[%case "Trigger Below X":%]
			[%actionName = activatesIfLessThanValue(action.Name,
actionNumber);%]
			[%constantStrings = createConstantAction(action.Value,
actionName, actionNumber, posXActions, posYActions);%]
		[%case "Trigger Above X":%]
			[%actionName = activatesIfGreaterThanValue(action.Name,
actionNumber);%]
			[%constantStrings = createConstantAction(action.Value,
actionName, actionNumber, posXActions, posYActions);%]
		[%case "Direct Percentage":%]
			[%actionName = directPercentage(action.Name, actionNumber);%]
			[%constantStrings = createConstantAction(action.Value,
actionName, actionNumber, posXActions, posYActions);%]
		[%case "Inverse Percentage":%]
			[%actionName = inversePercentage(action.Name, actionNumber);%]
			[%constantStrings = createConstantAction(action.Value,
actionName, actionNumber, posXActions, posYActions);%]
	[%}%]

	set_param([sys '/[%=actionName%]'],'Position', [[%=posXActions%]
[%=posYActions%] [%=(posXActions+width)%] [%=(posYActions+height)%]]);
	[%posYActions = posYActions + height + 30;%]

	[%actionsChartMap.put(actionName, constantStrings);%]
	[%actionNumber = actionNumber+1;%]
[% } %]

[% for (decisor in container.hasComponents.select(i|i.isKindOf(Decisor))) {%]

	[%if(decisor.Name == null){
		decisor.Name = "Decisor" + decisorNumber;
	}
	else{
		decisor.Name = decisor.Name + decisorNumber;
	}%]

	[% for (connection in
container.hasComponents.select(i|i.isKindOf(DecisorConnection))){%]
		[%if(connection.Output.Name.equals(decisor.Name)){%]
			[%countDecisorInputs = countDecisorInputs + 1; %]
		[%}%]
	[%}%]
```

```
	[%switch (decisor.DecisorType){%]
		[%case "Priority Decisor":%]
			[%decisorName = priorityDecisor(decisor.Name, decisorNumber,
posXDecisors, posYDecisors);%]
		[%case "Sequential Decisor":%]
			[%decisorName = sequentialDecisor(decisor.Name, decisorNumber,
countDecisorInputs, decisor.ConnectionNumber, posXDecisors, posYDecisors);%]
		[%case "Join Decisor":%]
			[%decisorName = joinDecisor(decisor.Name, decisorNumber,
countDecisorInputs, posXDecisors, posYDecisors);%]
	[%}%]

	set_param([sys '/[%=decisorName%]'],'Position', [[%=posXDecisors%]
[%=posYDecisors%] [%=(posXDecisors+width)%] [%=(posYDecisors+height)%]]);
	[%posYDecisors = posYDecisors + height + 30;%]
	[%decisorNumber = decisorNumber+1;%]
	[%countDecisorInputs = 0;%]
[% } %]

[% for (actuator in container.hasComponents.select(i|i.isKindOf(Actuator))) {%]
	[%switch (actuator.ActuatorType){%]
		[%case "On/Off Light":%]
			[%=onOffLightBulb(actuator.Name, posXActuators,
posYActuators)%]
		[%case "Dimmer Light":%]
			[%=dimmerLightBulb(actuator.Name, posXActuators,
posYActuators)%]
		[%case "Heater":%]
			[%=heaterState(actuator.Name, posXActuators, posYActuators)%]
		[%case "Blinds":%]
			[%=blinds(actuator.Name, posXActuators, posYActuators)%]
		[%case "Lock":%]
			[%=lock(actuator.Name, posXActuators, posYActuators)%]
	[%}%]

	[%posYActuators = posYActuators + height + 30;%]
[% } %]

[% for (connection in
container.hasComponents.select(i|i.isKindOf(StandardConnection))){%]
	[%if(connection.caseType.value == 0){%]

	add_line(blockDiagram,'[%=connection.Input.Name%]/2','[%=connection.Output.Na
me%]/1','autorouting','on');
	[%}else{%]

	add_line(blockDiagram,'[%=connection.Input.Name%]/1','[%=connection.Output.Na
me%]/1','autorouting','on');
	[%}%]
%(--[%=connection.Input.Name%],[%=connection.Output.Name%])
[%}%]

[% for (connection in
container.hasComponents.select(i|i.isKindOf(DecisorConnection))){%]
	add_line(blockDiagram,'[%=connection.Input.Name%]/1','[%=connection.Output.Na
me%]/[%=connection.Priority%]','autorouting','on');
%([%=connection.Input.Name%],[%=connection.Output.Name%])
[%}%]
```

```
[% for (connection in
container.hasComponents.select(i|i.isKindOf(StandardConnection)).select(i|i.Input.isK
indOf(Sensor))){
		var action = connection.Output;
		var subSystemArray = "Simulink.BlockDiagram.createSubSystem([";%]

		[%while(action.isKindOf(Action)){%]
			[%subSystemArray = subSystemArray.concat("get_param('" +
container.Name + "/" + action.Name + "', 'handle') ");%]

			[% for (constant in actionsChartMap.get(action.Name)){%]
				[%subSystemArray = subSystemArray.concat("get_param('"
+ container.Name + "/" + constant + "', 'handle') ");%]
			[%}%]

			[%action =
container.hasComponents.select(i|i.isKindOf(StandardConnection)).select(k|k.Input.Nam
e.equals(action.Name)).Output.first();%]
		[%}%]

		[%subSystemArray = subSystemArray.concat("])");%]
		[%=subSystemArray%]
[%}%]

[%
actionNumber = 0;
decisorNumber = 0;
countDecisorInputs = 0;
%]
[%}%]
```

9.8.2 Execution code

Listing 12 – Domatica_Sensor_Generation.egl

```
[%
import "Sensors/Domatica_PresenceSensor_Generation.egl";
import "Sensors/Domatica_DaylightSensor_Generation.egl";
import "Sensors/Domatica_PushButton_Generation.egl";
import "Sensors/Domatica_TimeSensor_Generation.egl";
import "Domatica_Action_Generation.egl";
%]

[% operation createSensorsUserParameters(container:Container){
	for (sensor in container.hasComponents.select(i|i.isKindOf(Sensor))) {
		switch (sensor.SensorType){
			case "Presence Sensor":
				presenceSensorUserParameters(container.Name, sensor);
			case "Temperature Sensor":
				temperatureSensorUserParameters(container.Name,
sensor);
			case "Daylight Sensor":
				daylightSensorUserParameters(container.Name, sensor);
			case "Push Button Sensor":
				pushButtonUserParameters(container.Name, sensor);
			case "Time Sensor":
				timeSensorUserParameters(container.Name, sensor);
		}
	}
} %]
```

```
[% operation setSensorsInitialState(container:Container){
	for (sensor in container.hasComponents.select(i|i.isKindOf(Sensor))) {
		switch (sensor.SensorType){
			case "Presence Sensor":
				presenceSensorInitialState(container.Name, sensor);
			case "Temperature Sensor":
				temperatureSensorInitialState(container.Name, sensor);
			case "Daylight Sensor":
				daylightSensorInitialState(container.Name, sensor);
			case "Push Button Sensor":
				pushButtonInitialState(container.Name, sensor);
			case "Time Sensor":
				timeSensorInitialState(container.Name, sensor);
		}
	}
} %]

[% operation createSensorsUserTasks(container:Container){
	for (sensor in container.hasComponents.select(i|i.isKindOf(Sensor))) {
		switch (sensor.SensorType){
			case "Presence Sensor":
				presenceSensorUserTasks(container.Name, sensor);
			case "Temperature Sensor":
				temperatureSensorUserTasks(container.Name, sensor);
			case "Daylight Sensor":
				daylightSensorUserTasks(container.Name, sensor);
			case "Push Button Sensor":
				pushButtonUserTasks(container.Name, sensor);
			case "Time Sensor":
				timeSensorUserTasks(container.Name, sensor);
		}
	}
} %]

[% operation createSensorEvents(containerName:String, elementList : Sequence,
isPositive: Boolean){
	for (sensor in elementList) {
		if(sensor.isKindOf(Sensor)){
			switch (sensor.SensorType){
				case "Presence Sensor":
					presenceSensorEvents(containerName, sensor,
isPositive);
				case "Temperature Sensor":
					temperatureSensorEvents(containerName, sensor,
isPositive);
				case "Daylight Sensor":
					daylightSensorEvents(containerName, sensor,
isPositive);
				case "Push Button Sensor":
					pushButtonEvents(containerName, sensor,
isPositive);
				case "Time Sensor":
					timeSensorEvents(containerName, sensor,
isPositive);
			}
		}
	}
} %]
```

```
[% operation createSensorVerification(containerName:String, elementList : Sequence,
isPositive: Boolean){
	for (sensor in elementList) {
		if(sensor.isKindOf(Sensor)){
			switch (sensor.SensorType){
				case "Presence Sensor":
					presenceSensorVerification(containerName,
sensor, isPositive);
				case "Temperature Sensor":
					temperatureSensorVerification(containerName,
sensor, isPositive);
				case "Daylight Sensor":
					daylightSensorVerification(containerName,
sensor, isPositive);
				case "Push Button Sensor":
					pushButtonVerification(containerName, sensor,
isPositive);
				case "Time Sensor":
					timeSensorVerification(containerName, sensor,
isPositive);
			}
		}
	}
} %]

[% operation getValueOfSensor(containerName:String, sensor : Sensor):String{
	switch (sensor.SensorType){
		case "Presence Sensor":
			return getValueOfPresenceSensor(containerName, sensor);
		case "Temperature Sensor":
			return getValueOfTemperatureSensor(containerName, sensor);
		case "Daylight Sensor":
			return getValueOfDaylightSensor(containerName, sensor);
		case "Push Button Sensor":
			return getValueOfPushButton(containerName, sensor);
		case "Time Sensor":
			return getValueOfTimeSensor(containerName, sensor);
	}
}%]

[% operation getConnectionCaseType(sensor : Sensor):Boolean{
	var connection = StandardConnection.allInstances().select(i|i.Input = sensor
and not(i == null));

	if(connection.caseType.first().value == 1){
		return true;
	}

	return false;
}%]

[% operation getValueFromBoolean(v:Boolean):Integer{
	if(v){
		return 1;
	}

	return 0;
}%]
```

```
[% operation hasPositiveCase(sensor:PresenceSensor): Sensor{

	var s = hasPairCase(sensor);

	if(not (s == null) and getConnectionCaseType(s)){
		return s;
	}
	return null;
}%]

[% operation hasNegativeCase(sensor:PresenceSensor): Sensor{

	var s = hasPairCase(sensor);

	if(not (s == null) and (not getConnectionCaseType(s))){
		return s;
	}
	return null;
}%]

[% operation hasPairCase(sensor:PresenceSensor): Sensor{
	var joinConnection = Connection.allInstances().select(i|i.Input ==
sensor).first();

	while((not joinConnection.Output.isKindOf(JoinDecisor)) and (not
joinConnection.Output.isKindOf(Actuator))){
		joinConnection = Connection.allInstances().select(i|i.Input ==
joinConnection.Output).first();
	}

	if(joinConnection.Output.isKindOf(Actuator)){
		return null;
	}

	var connectionAux = Connection.allInstances().select(i|i.Output ==
joinConnection.Output).select(j| not (j == joinConnection)).first();


	while(not connectionAux.Input.isKindOf(Sensor)){
		connectionAux = Connection.allInstances().select(i|i.Output ==
connectionAux.Input).first();
	}

	if(sensor.ID == connectionAux.Input.ID){
		return connectionAux.Input;
	}

	return null;
} %]
```

Listing 13 – Domatica_templates.egl

```
[%import "Domatica_Sensor_Generation.egl";
import "Domatica_Actuator_Generation.egl";
import "Domatica_Decisor_Generation.egl";
import "Domatica_Action_Generation.egl";
import "Domatica_Utils.egl";
%]

[% operation initXMLFile(){%]
<?xml version="1.0" encoding="UTF-8"?>
<iDomUserProgram>
[% } %]
```

```
[% operation initUserParameters(){%]
	<UserParameters>
[% } %]

[% operation initUserTasks(){%]
	<UserTasks>
[% } %]

[% operation closeUserParameters(){%]
	</UserParameters>
[% } %]

[% operation closeUserTasks(){%]
	</UserTasks>
[% } %]

[% operation closeXMLFile(){%]
</iDomUserProgram>
[% } %]

[% operation createUserParameters(container : Container){
	createDefaultUserParameters();
	createSensorsUserParameters(container);
	createDecisorsUserParameters(container);
	createActionsUserParameters(container);
} %]

[% operation createUserTasks(container : Container){
	createDefaultUserTasks(container);
	createSensorsUserTasks(container);
	createActionsUserTasks(container);
	createDecisorsUserTasks(container);
	createActuatorUserTasks(container);
} %]

[% operation createEvents(containerName: String, elementList : Sequence, isPositive:
Boolean){
	createSensorEvents(containerName, elementList, isPositive);
	createDecisorEvents(containerName, elementList, isPositive);
} %]

[% operation createVerification(containerName: String, elementList : Sequence,
isPositive: Boolean){
	createSensorVerification(containerName, elementList, isPositive);
	createDecisorVerification(containerName, elementList, isPositive);
} %]

[% operation getValueOfElement(containerName: String, element : Element):String{
	if(element.isKindOf(Sensor)){
		return getValueOfSensor(containerName, element);
	}

	if(element.isKindOf(Decisor)){
		return getValueOfDecisor(containerName, element);
	}
}%]
```

Listing 14 – XML_Generation.egl

```
[%
import "Domatica_templates.egl";
%]

[% operation generateXMLFile(){

	var newSensorNames = duplicateSensors();%]

	[% var container = Container.allInstances().first();%]

	[%=initXMLFile()%]

	[%=initUserParameters()%]
	[%=createUserParameters(container)%]
	[%=closeUserParameters()%]

	[%=initUserTasks()%]
	[%=createUserTasks(container)%]
	[%=closeUserTasks()%]

	[%=closeXMLFile()%]

[%
	deleteCreatedSensors(newSensorNames);
}%]

[% operation duplicateSensors(): Sequence{
	var newSensors = Sequence{};
	var connections2 =
StandardConnection.allInstances().select(i|i.Input.isKindOf(Sensor));
	var nConnections = connections2.size();
	var nSensors = Sensor.allInstances().size();
	var container = Container.allInstances().first();

	if(not (nConnections == nSensors)){
		for(c in Sensor.allInstances()){
			var connections =
StandardConnection.allInstances().select(i|i.Input == c);
			if(connections.size() > 1){
				var i = 1;
				var sequence = Sequence{c};
				var seqOfSensors = Sequence{};
				while(i < connections.size()){
					var newSensor = createSensor(c, i);
					seqOfSensors.add(newSensor);
					container.hasComponents.add(newSensor);

					connections.at(i).Input = newSensor;
					i = i+1;
				}
				sequence.add(seqOfSensors);
				newSensors.add(sequence);
			}
		}
	}
	return newSensors;
}%]
```

```
[% operation createSensor(sensor:Sensor, number:Integer):Sensor{
	var newSensor;

	switch (sensor.SensorType){
		case "Presence Sensor":
			newSensor = PresenceSensor.createInstance();
			newSensor.Timeout = sensor.Timeout;
		case "Temperature Sensor":
			newSensor = TemperatureSensor.createInstance();
		case "Daylight Sensor":
			newSensor = DaylightSensor.createInstance();
		case "Push Button Sensor":
			newSensor = PushButton.createInstance();
	}

	newSensor.ID = sensor.ID;
	newSensor.Name = sensor.Name + "_" + number;

	return newSensor;
} %]

[% operation deleteCreatedSensors(newSensors:Sequence){
	var container = Container.allInstances().first();

	for(seq in newSensors){
		for(s in seq.second()){
			var connection =
StandardConnection.allInstances().select(i|i.Input == s).first();
			connection.Input = seq.first();
			delete s;
		}
	}
}%]
```

9.9 HABITATION DSL

This section has the catalogues used by the author of this DSL to organize devices and functionalities.

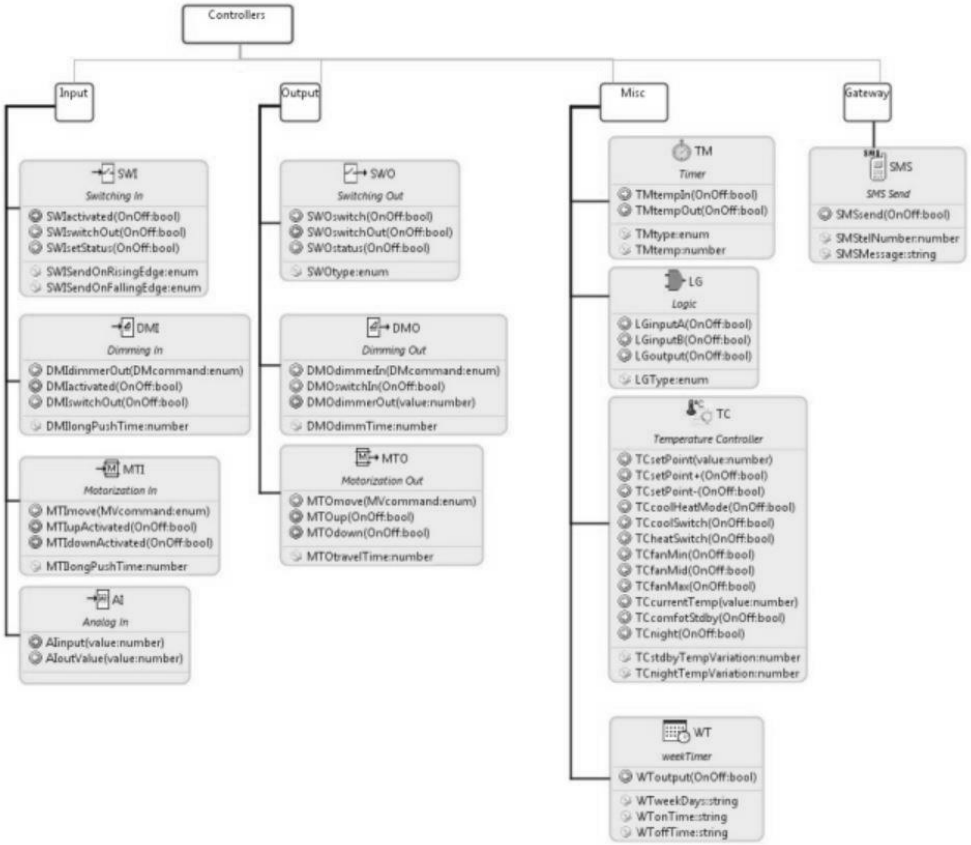


Figure 51 – Catalogue of Services, taken from [36]

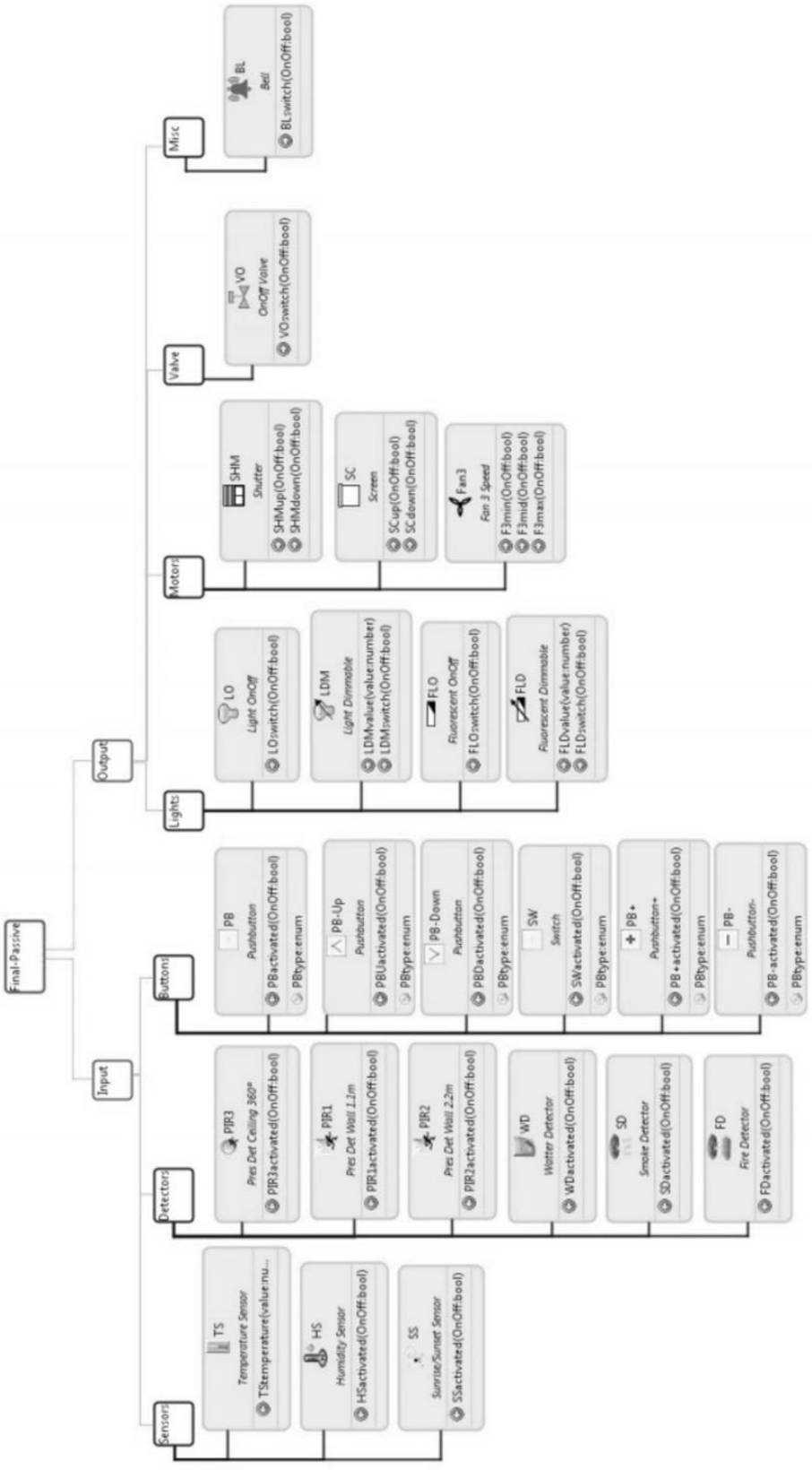


Figure 52 – Catalogue of Functional Units, taken from [36]

9.10 QUESTIONNAIRE

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

(Please do not fill these fields)
Date: _____
Number: _____

A DSL for Home Automation [Part I]

Thank you for your participation, your feedback is important for us. Through this questionnaire, your answers will be helpful in enhancing our language. Your response will be used only for the purpose of this project.

Don't forget we are validating a tool, therefore, there is no wrong answers.

General Information

Age: _____

Sex: ☐ Male ☐ Female

Country: _____

Education

Education Level:

☐ Basic ☐ Secondary ☐ Bachelor
☐ Master ☐ Doctoral

What is your field of study?

_____

Experience

Did you ever used hardware prototyping platforms, like Arduino and RaspberryPI?

Yes ☐ No ☐

How would you rate your experience with Simulation software (LabView, Simulink, ...)?

None ☐ ☐ ☐ ☐ Expert

How would you rate your experience using Home Automation equipment (LonWorks, KNX, X10, ...)?

None ☐ ☐ ☐ ☐ Expert

Page 1 of 3

Figure 53 – Questionnaire, Part I, Page 1

How would you rate your experience using Home Automation software (iDom, 4ETS, HAbitATION, ...)?

None ☐ ☐ ☐ ☐ Expert

Icons Validation

For each of the following icon, how would you rate the meaning that is associated to it, according to the description?

(Please circle the smiley that you consider most appropriate)

Palette Group	Icon ID	Icon	Icon Name	Icon Description	Immediate Association	Logical Association	No Association
Object	1		Container	Used to build the domotic behavior			
	2		Execution	Indicates that the behavior will be executed in the target platform			
	3		Schedule	A date can be defined for the behavior to run			
	4		Simulation	Indicates that the behavior will be simulated in the target platform			
Connection	5		Container Connection	Connects the Container to the other Objects			
	6		Connection to Decisors	Connects elements to Decisors. The order of the connections must begin in 1 with increments of 1 unit			
	7		Global Connection	Connects the elements: Sensors, Actions and Actuators			
Sensors	8		Daylight Sensor	Measures and provides the light intensity in lux			
	9		Temperature Sensor	Measures and provides the temperature in degrees Celsius			
	10		Presence Sensor	Indicates if there is presence in the room, through movement			
	11		Push Button	When pushed it activates/deactivates a behavior. If pressed, the value is given in increments of 10 units			
	12		Time Sensor	Used to limit the usage of certain Sensors between two distinct hours			

Page 2 of 3

Figure 54 – Questionnaire, Part I, Page 2

Actuators	13		Heater	A simple Heater that could be turned On or Off			
	14		On/Off	A type of light bulb that could only be turned On or Off			
	15		Dimmer	A type of light bulb that supports light intensity variation			
	16		Blinds	Blinds that are motorized and work with percentage, to define the openness level			
	17		Lock	A type of lock that can be Locked or Unlocked			
Actions	18		Trigger Below X	Indicates that the input value is below the defined Value			
	19		Trigger Above X	Indicates that the input value is above the defined Value			
	20		Trigger With X	The output is the defined Value when activated, otherwise has the value 0			
	21		Step Until X	The received base value is modified Step-by-Step after a certain Time, until it reaches the Target value			
	22		Direct Percentage	The Value corresponds to 100%, so the output is the direct percentage representation of the input			
	23		Inverse Percentage	The Value corresponds to 0%, so the output is the inverse percentage representation of the input			
Decisors	24		Priority Decisor	Receives 2 connections, where the highest priority overrules the other			
	25		Sequential Decisor	Supports multiple connections, which are validated by priority			
	26		Join Decisor	Supports multiple connections, where there are no priority rules			

Page 3 of 3

Figure 55 – Questionnaire, Part I, Page 3

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

(Please do not fill these fields)
Date: _____

Number: _____

A DSL for Home Automation [Part II]

Thank you for your participation, your feedback is important for us. Through this questionnaire, your answers will be helpful in enhancing our language. Your response will be used only for the purpose of this project.

Don't forget we are validating a tool, therefore, there is no wrong answers.

Tasks to perform

In this section, we challenge you to use our tool to create some automated behaviors for this house:

Area ID	Area Name	Element	Element ID	Element Type
1	Garage	Sensor	PS_G	Presence Sensor
		Actuator	SL_G	On/Off Light
2	Hall	Sensor	PS_H	Presence Sensor
		Actuator	SL_H	On/Off Light
3	Living room	Sensor	DLS_LR	Daylight Sensor
			TS_LR	Temperature Sensor
		Actuator	B_LV	Blinds
			H_LR	Heater
4	Kitchen	Sensor	PS_K	Presence Sensor
		Actuator	SL_K	On/Off Light
5	Master bedroom	Sensor	PB_B1	Push Button
			DLS_B1	Daylight Sensor
			PS_B1	Presence Sensor
		Actuator	DL_B1	Dimmer Light

You should use the same names as the ones of the floor plant. You are free to choose the deployment type (Execution/Simulation).

Task 1 – "Isabella wants the light (DL_B1) in her bedroom, to be lit up to a value of 60 when presence is detected in that space." (Please consider that a Dimmer Light supports values between 0 and 100)

How would you rate the difficulty on developing this task?

Easy ☐ ☐ ☐ ☐ Hard

How certain are you that the task was completed successfully?

0-20% ☐ 20-40% ☐ 40-60% ☐ 60-80% ☐ 80-100% ☐

Page 1 of 5

Figure 56 – Questionnaire, Part II, Page 1

(This next task is an evolution of the previous one, therefore, the same elements will be used. Please, do not copy them from the other files)

Task 2 – "Isabella figured that having the light on during the day is not necessary. Therefore, she only wants her bedroom light (DL_B1) to be on when presence is detected and the daylight is lower than 100 lux (SI unit for measuring the light intensity)." (Please consider that an Actuator only supports 1 input)

How would you rate the difficulty on developing this task?

Easy ☐ ☐ ☐ ☐ Hard

How certain are you that the task was completed successfully?

0-20% ☐ 20-40% ☐ 40-60% ☐ 60-80% ☐ 80-100% ☐

(This next task is an evolution of the previous one, therefore, the same elements will be used. Please, do not copy them from the other files)

Task 3 – "Isabella sometimes needs the bedroom light (DL_B1) to be on during the day. Thus, she would like to have control over the automatic behavior through the push button, which when pressed would set the bedroom light level at 100." (Please consider that an Actuator only supports 1 input)

How would you rate the difficulty on developing this task?

Easy ☐ ☐ ☐ ☐ Hard

How certain are you that the task was completed successfully?

0-20% ☐ 20-40% ☐ 40-60% ☐ 60-80% ☐ 80-100% ☐

Automatic Behaviors

Using this tool, do you think that would be possible to turn the light of the kitchen on when you enter in the garage?

☐ Yes
☐ No. Why? _____

Using this tool, do you think that would be possible to schedule a behavior to occur only on the weekend?

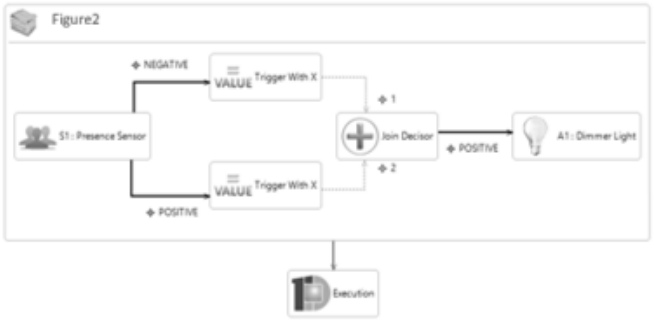
☐ Yes
☐ No. Why? _____

Page 2 of 5

Figure 57 – Questionnaire, Part II, Page 2

Behavior Identification

The following images represents two different automatic behavior.



What the user wants to automate in the Figure 1?

What the user wants to automate in the Figure 2?

Page 3 of 5

Figure 58 – Questionnaire, Part II, Page 3

General Opinion

Which one of the previous tasks was the most difficult to perform?

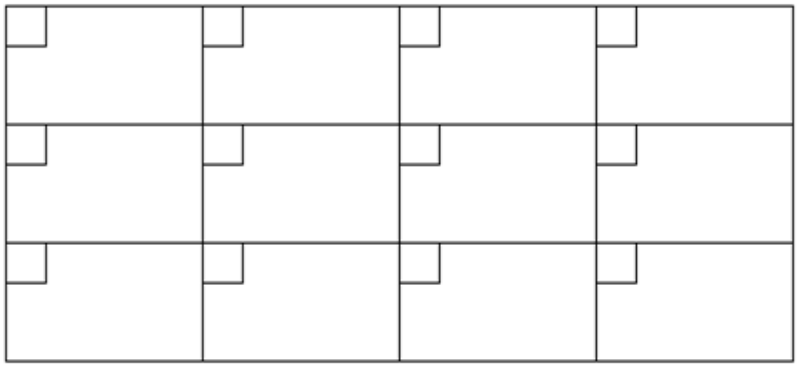
☐ None ☐ Task 1 ☐ Task 2 ☐ Task 3

Would you use this tool?

☐ Yes ☐ No. Why? _____

Which are the icons that you considered poorly chosen?

(Use the small square to indicate the icon number presented on the icon table and use the box to draw/describe a better option)



Additional Feedback

Please list any ideas in which our language could be improved.

Page 4 of 5

Figure 59 – Questionnaire, Part II, Page 4

Personal Information

Providing the following information is optional.

First Name _____ Last Name _____

Email _____

Would you like someone to contact you regarding your responses on this survey?

☐ Yes | ☐ No

Thank you for taking the time to fill out our questionnaire. Your input is greatly appreciated.

Notes

(Please do not fill this space)

Page 5 of 5

Figure 60 – Questionnaire, Part II, Page 5