User-friendly Home Automation using IDP

Thijs ALENS

Promotor:	S. Vandevelde	Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: Master in de industriële wetenschappen elektronica-ICT
Supervisor:	L. Van Laer	

Academic year 2024 - 2025

Acknowledgement

I would like to thank my Promotor, S. Vandevelde, and supervisor L. Van Laer, for their guidance and support throughout this thesis. Their expertise and insights have been invaluable in shaping the direction of this work. I would also like to thank my family and friends who supported me throughout the journey of my studies and this thesis.

Samenvatting

Een domotica dient om een woning te automatiseren. Dit kan variëren van heel eenvoudige taken, zoals een licht inschakelen wanneer een schakelaar wordt omgezet, tot complexere taken zoals het regelen van de temperatuur — en daarbij ook de ramen, verwarming en gordijnen — afhankelijk van allerlei factoren. Het opzetten van dergelijke automatisaties is echter niet zo vanzelfsprekend: de gebruiker moet enige kennis hebben van domotica en eerdere ervaring hebben met programmeren. De meeste mensen beschikken niet over deze kennis en kunnen daardoor niet ten volle profiteren van de voordelen die domotica biedt.

Wij ontwikkelden een gebruiksvriendelijke applicatie die het voor iedereen mogelijk moet maken om hun huis te automatiseren, zonder dat daarvoor diepgaande technische kennis vereist is. We deden dit door bestaande domotica-systemen onder de loep te nemen en te onderzoeken waar verbeteringen mogelijk waren. We maakten gebruik van FO(·), een taal gebaseerd op eerste-orde logica. Deze taal kan worden gebruikt met IDP-Z3, die redeneert over de in FO(·) beschreven kennis om verschillende taken uit te voeren. De FO(·)-taal is echter, net als andere talen in de wereld van domotica, niet eenvoudig te begrijpen of te gebruiken. Daarom onderzochten we mogelijke alternatieven voor FO(·), waaruit bleek dat een blocks-based editor een goede optie zou zijn.

Vervolgens ontwikkelden we een applicatie die gebruikmaakt van zelfgemaakte blokjes om automatisaties te creëren. De blokjes worden vertaald naar een FO(·)-beschrijving, die vervolgens door IDP-Z3 gebruikt kan worden om de automatisaties uit te voeren. We besteedden daarbij bijzondere aandacht aan de gebruiksvriendelijkheid van de applicatie, omdat we willen dat iedereen ermee aan de slag kan.

Om dit te testen voerden we een gebruikerstest uit met 14 proefpersonen. Hoewel de resultaten van deze test met enige voorzichtigheid geïnterpreteerd moeten worden, wijzen ze erop dat de applicatie gebruiksvriendelijk is. De testpersonen gaven aan dat ze zich, met behulp van de applicatie, redelijk zelfzeker voelen in hun vermogen om hun huis te automatiseren. Deze applicatie is echter nog niet af; er zijn nog vele mogelijkheden om hierop verder te bouwen, bijvoorbeeld een integratie met bestaande domotica-systemen of verdere uitwerking van de blokjes.

Summary

A home automation system is designed to automate a house. This can range from very simple tasks, such as turning on a light when a switch is flipped, to more complex tasks like regulating the temperature — and with that, controlling windows, heating and curtains — depending on various factors. However, setting up such automations is not always straightforward. The user needs to be somewhat familiar with home automation and have basic programming skills. Most people lack this knowledge and therefore cannot fully enjoy the benefits that home automation has to offer.

We developed a user-friendly application that aims to make it possible for anyone to automate their home, without requiring in-depth technical knowledge. We did this by analyzing existing home automation systems and identifying areas for improvement. In this process, we used FO(·), a language based on first-order logic. This language can be used with the reasoning engine IDP-Z3, which reasons over the knowledge described in FO(·) to perform various tasks. However, the FO(·) language — like many others in the home automation field — is not easy to understand or use. For this reason, we conducted a study of possible alternatives to FO(·), which showed that a blocks-based editor could be a good solution.

We then developed an application that uses custom-made blocks to create automations. These blocks are translated into a FO(·) description, which can then be used by IDP-Z3 to execute the automations. We paid particular attention to the application's user-friendliness, as our goal was to make it accessible to everyone.

To test this, we conducted a user study with 14 participants. Although the results should be taken with a grain of salt, they indicate that the application is user-friendly. Participants reported feeling reasonably confident in their ability to automate their home using the application. However, this application is not yet complete; there are still many possibilities for further development, such as integration with existing home automation systems or further refinement of the blocks.

Contents

Chapter 1

Introduction

1.1 Positioning

We are living in a world where automation is adopted at a rapidly increasing pace. From automatically sorting luggage to the right plane at the airport, to fully self-driving cars, automation is becoming an integral part of our lives. This evolution, paired with the emergence of the Internet of Things (IoT), where many devices have become "smart" and automation-ready, has not only revolutionized industries but also opened up new possibilities for homeowners.

In essence, home automation is a way to automate tasks in a home. These can be simple tasks such as automatically turning on lights upon entering a room, to more advanced operations like regulating home climate. This has the potential to significantly streamline daily routines, reduce costs, and improve overall comfort.

One way to configure a home environment is by using a home automation system that runs on a local server in conjunction with Home Assistant (HA). HA is an application that integrates various devices from different brands into a single functional app. It also provides a framework through which users can automate their homes.

A home automation system is essentially a set of rules that defines the behavior of the home, making it well-suited for a knowledge-based system. Such a system requires the user to input a set of rules, which are then processed by a reasoning engine to make the necessary decisions.

IDP-Z3 is an example of such a reasoning engine, utilizing a formal description to define these rules. One of the key advantages of IDP-Z3 is its ability to separate knowledge from its application. This separation allows users to focus on defining the rules without needing to program the decision-making process themselves, as the reasoning engine handles this automatically.

1.2 Problem statement

One of the main challenges to home automation is the steep learning curve associated with setting up and maintaining home automation systems. If users wish to implement their own home automation, they can use home assistants user interface (UI). Automations, however, require that the user has a basic understanding of programming and is familiar with specific technical tools. This way they can manually modify the configuration files and troubleshoot any issues that may be presented, which is not easily done by non-experts.

IDP-Z3 can make this process simpler by making use of a declarative knowledge base, avoiding the need to program automations. However, the IDP-Z3 language itself is not particularly user-friendly for non-experts and can become quite complex, especially when dealing with intricate rules or scenarios. When users are confronted with a large problem domain, it can be difficult to maintain an overview and make necessary adjustments. It also has a lot of possibilities that are not useful in the context of home automation, which can be confusing or overwelming.

1.3 Objectives

This thesis seeks to address this issue by investigating how IDP-Z3 can be utilized to make home automation more accessible. Instead of needing to create automations, triggers, and aligning them with each other, the user only needs to describe the desired behavior of the home in a declarative language. A graphical user interface (GUI) that provides structure, along with a more limited subset of the IDP language, could help make IDP-Z3 a more viable option for home automation.
The primary research question addressed in this thesis is:
How do we design an IDP-Z3 framework that enables end-users to automate their homes in a user-friendly way?
This overarching question is further explored through the following sub-questions:

- What is the optimal user-friendly interface for addressing this problem?
- Which subset of the IDP-language is needed to configure a home?

The objectives of this thesis are:

- Decide on a subset of the IDP-language that has all the functionality needed for home automation.
- Design a user-friendly UI that is most suited for IDP in combination with home automation.

Chapter 2

Literature review

In this chapter, we discuss existing research relevant to the topics covered in this thesis. First, we elaborate on home automation, with a brief overview of HA and how users can configure their homes using it. Next, we will explore the basics of FO(·) and IDP-Z3. After that, we will examine existing user-friendly FO(·) alternatives, highlighting their strengths and weaknesses in the context of this use case. To close we will discuss the state of the art and how we can learn from it.

2.1 Home automation

Home automation is a broad term used when discussing the automation of a home. It ranges from simple tasks, such as automatically turning on a light when entering a room, to complex tasks, like adjusting the house temperature based on various variables. It also refers to home security: when should the security camera automatically record, when should the doors automatically lock, what should happen if the alarm goes off, etc. Automating tasks, like automatically making coffee at the start of the day, is also considered part of home automation. Additionally, it can help manage energy consumption throughout the day to reduce costs.

Home automation consists of 3 main parts:

- Smart home devices
- A smart hub/server
- An application

Smart home devices A smart home device (or smart device) can be either of two things: a sensor or an actuator. A sensor can detect an event, while an actuator can respond to a trigger. A simple example is a motion detector (sensor) that automatically turns on a light (actuator) when there is movement (trigger). Depending on the device, there may be some additional (smart) features. For example, the light could have an internal clock that provides the current time. This allows the light to automatically adjust its brightness based on the expected amount of natural light at that time. In summary, smart devices are the physical hardware (sensors and actuators) combined with software used for the communication between devices, which, in most cases, do not contain any smart home logic themselves.

Smart hub/server The smart hub is a central device that connects the complex hardware of smart devices to the user. Its function is to receive data from sensors and send commands to actuators, serving as a central hub for the devices, so to speak. This hub can be provided by a manufacturer specifically for their smart devices, or it can be a generic one designed to be compatible with as many devices as possible.

Application The application allows the user to configure their home. Most often, this is done through a graphical user interface (GUI) where the user can create automations, view the status of devices, monitor active automations, easily communicate with the devices, etc.

2.1.1 Home assistant

Home assistant (HA) [1] is an open-source, all-in-one application for home automation. In a home environment, it typically runs on a local server, where smart devices can be connected and configured. Because it is local, it provides strong security for user data. HA has a large and growing community, which ensures compatibility with a wide range of devices and brands, eliminating the need for multiple apps to control a home environment.
HA also offers multiple UIs (known as dashboards) where users can monitor the state of their home. Users can select a dashboard created by others, design one themselves, or build on an existing dashboard. They can interact with their devices directly through these dashboards. Additionally, users can create automations using a separate UI, which are actions executed when a set trigger occurs. Fully understanding how these automations work requires a deeper understanding of how HA functions. In the following paragraphs we will discuss some terminology used inside HA. An overview of these concepts is shown in Figure 2.1.

Entities These are the lowest level possible. They represent single sensors/actuators like a temperature sensor, a lightswitch, a light, etc.

Devices These are a group of entities. It could be that a device has 1 entity (e.g. a lightswitch), but it also could have multiple entities (e.g. a motion sensor that is also capable of capturing the temperature).

Areas These are groups of devices that could correspond to rooms in a house. For instance, the living room could have devices like a light-switch, a motion sensor (that detects if someone

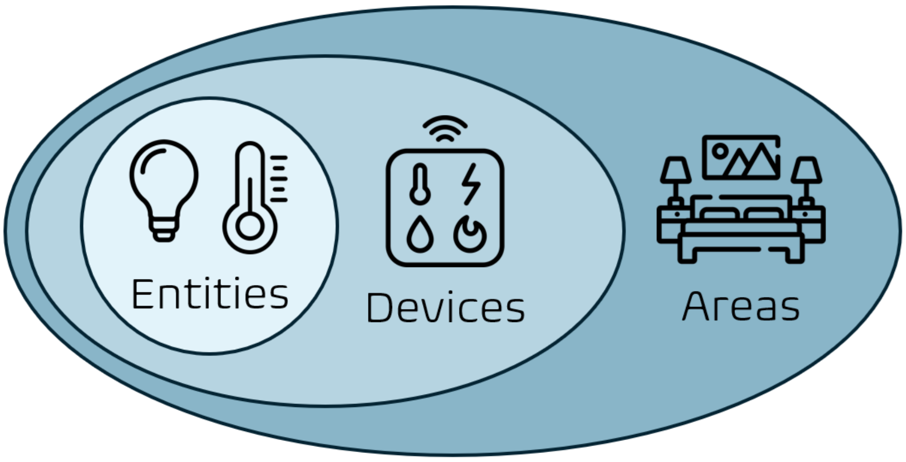


Figure 2.1: The main hierarchy of HA [1]

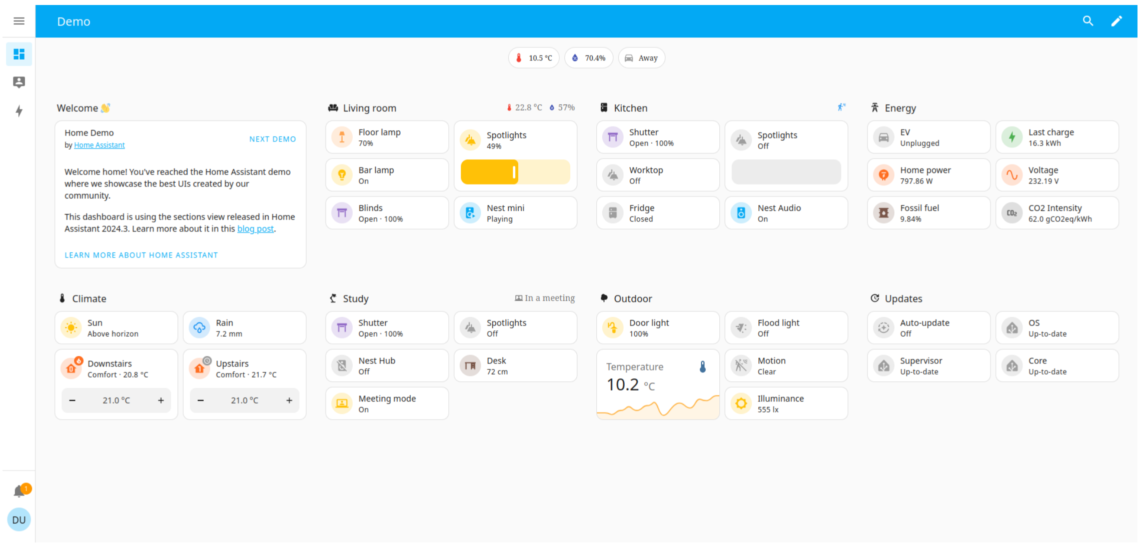
is in the room), a set of speakers, etc. All of these devices could be grouped together in one area.

Scenes Scenes are used to automatically set a set of devices to a predefined state. This could, for example, pre-configure the actuators of the living room to watch a movie. The user could set up a scene where, if activated, all the lights dim, the tv turns on and all the blinds in the living room close.

Automations Automations are used to automate things. These automations consist of three things. First a "trigger" needs to happen to activate the automation. This could, for example, be a motion sensor detecting movement. After the trigger, possible "conditions" are checked. These are additional requirements that need to be met before executing the automation, for example, ensuring someone is home. Finally "actions" are performed, these are the outcomes that occur after a trigger and when the conditions are met, like turning on the light in a room.

Scripts Scripts are predefined actions that are usable in automations. For example, a user may create a script to turn on all the lights in a room. This script can then be utilized in an automation that turns on the lights when someone enters the room. If the user later wants to create another automation to turn on the lights when the carport opens, they can reuse the same script. The benefit of scripts is their maintainability: if a new light is added to the room, the user only needs to update the script. Both automations will automatically use the updated script, eliminating the need to modify each automation individually.

[1] Icons by `https://www.flaticon.com/free-icons/pixel`


Demo

10.5 °C
70.4%
Away

Welcome

Home Demo
by Home Assistant
NEXT DEMO

Welcome home! You've reached the Home Assistant demo where we showcase the best UIs created by our community.

This dashboard is using the sections view released in Home Assistant 2024.3. Learn more about it in this blog post.

LEARN MORE ABOUT HOME ASSISTANT

Living room 22.8 °C 57%

Floor lamp 70%
Spotlights 49%
Bar lamp On
Blinds Open · 100%
Nest mini Playing

Kitchen

Shutter Open · 100%
Spotlights Off
Worktop Off
Fridge Closed
Nest Audio On

Energy

EV Unplugged
Last charge 16.3 kWh
Home power 797.86 W
Voltage 232.19 V
Fossil fuel 9.84%
CO2 Intensity 62.0 gCO2eq/kWh

Climate

Sun Above horizon
Rain 7.2 mm
Downstairs Comfort · 20.8 °C
− 21.0 °C +
Upstairs Comfort · 21.7 °C
− 21.0 °C +

Study In a meeting

Shutter Open · 100%
Spotlights Off
Nest Hub Off
Desk 72 cm
Meeting mode On

Outdoor

Door light 100%
Flood light Off
Temperature 10.2 °C
Motion Clear
Illuminance 555 lx

Updates

Auto-update Off
OS Up-to-date
Supervisor Up-to-date
Core Up-to-date

DU

Figure 2.2: An example of a dashboard in HA

2.1.2 Home assistant's UI

As specified above, HA's UI has two parts. The dashboard, which is fully customizable by the user, and the automation editor, which is a pre-configured UI.

Dashboard The dashboard is the central interface where users control their smart home. It is primarily built using widgets, which can represent virtually anything the user envisions. For instance, a widget could represent a single room, containing sub-widgets to control each device within that room. Alternatively, a widget could display the entire house, allowing users to manage all devices from a single view. Users even have the option to create individual dashboards for each room if they wish. An example of a dashboard is found in Figure 2.2. This dashboard is used as one of the demo's on the HA website [2].

Automation editor The automation editor is designed for creating and editing automations. It follows a fixed structure: a trigger is specified first, followed by a condition, and concluded with an action.

This consistent structure offers a key advantage: users only need to learn one framework for creating automations. However, this approach has its drawbacks. Users still need to understand how devices should be utilized within this framework, which can be challenging without prior knowledge. Additionally, by managing automations across separate user interfaces, it becomes difficult to maintain a clear overview of what each automation is designed to do.

Table 2.1 Explanation of symbols used by FO(·)

FO(·) symbol	ASCII characters	meaning
$\wedge$	&	logical and
$\vee$	|	logical or
$\neg$	~	negation
$\Rightarrow$	=>	implication
$\Leftrightarrow$	<=>	equivalence
$\forall$	!	for all
$\exists$	?	for at least 1

2.1.3 Issues with the current automation configuration

While Home Assistant provides its users with an all in one application with a UI for basic home configuration, it does not allow for more complex rules. The user will still need to dive in the configuration files themself to control their devices so they do what is needed. These configuration files are YAML-files, which are clear for people with an IT background, but can be difficult to follow for non-expert people. Fortunately, HA has an active community that often helps less experienced users navigate these difficulties.

Despite this support, the community still struggles to address certain challenges. As highlighted in [3], users face three primary issues when modeling their homes in YAML: debugging (68%), implementation (27%), and optimization (5%). The paper evaluates six validation tools, including the HA IDE, and reports that they have a very high precision (75%-94% of the identified errors were actual errors). However, they suffer from very low recall. The tools only identified 9-11 bugs out of 129.

This indicates that while validation tools exist, they are not very effective in practice. Robust validation is essential for users, as they are likely to encounter bugs or problems that cannot be resolved through an IDE. This often forces them to edit YAML files directly, which is not ideal. Even worse, if mistakes are made in the YAML files, identifying and correcting them becomes nearly impossible without better tools or an active community. When designing a user-friendly application, validation should be kept in consideration.

2.2 FO(·)

FO(·)[2] (aka FO-dot) is the Knowledge Representation Language used by the IDP-Z3 reasoning engine [4]. It is an extension of first-order logic (FOL), which makes use of the logic operators described in Table 2.1.

[2]FO(·) is pronounced "Eff-Oh-dot"

An IDP-Z3 knowledge base consists of, at a minimum, a vocabulary and a theory. The vocabulary defines the problem domain and specifies the symbols that will be used in the theory. The theory consists of rules that apply to these symbols. The structure, which is optional, describes a single, specific situation. Here is a simple example to illustrate these blocks and the basics of FO(·), which will be explained further on.

```
vocabulary V {
    type Light
    type State := {On, Off, Dimmed}
    type SimpleState := {On, Off} <: State

    stateOfLight: Light → State
    brightnessLvl: Light → Int
    isSomeoneHome: () →  𝔹
}

theory T:V {
    {
       // A light is Off if the brightnesslevel is equal to 0
       ∀l in Light: stateOfLight(l) = Off ← brightnessLvl(l) = 0.

       // A light is On if the brightnesslevel is greater then 0
       ∀l in Light: stateOfLight(l) = On ← brightnessLvl(l) > 0.
    }

    // if nobody is home, then all the lights should be off
    ∀l in Light: ¬isSomeoneHome() ⇒ stateOfLight(l) = Off.

    // the brightnesslvl of all the lights should be divisable by
   10 and stay between 0 and 100
    ∀l in Light: (brightnessLvl(l) ≥ 0) ∧
    (brightnessLvl(l) ≤ 100) ∧ (brightnessLvl(l) % 10 = 0).

    // the light l should be off, if and only if nobody is home or
   the brightnessLvl is equal to 0
    ∀l in Light: stateOfLight(l) = Off ⇔
    (¬isSomeoneHome() ∨ (brightnessLvl(l) = 0)).
}

structure S:V {
    Light := {light1, light2, light3, light4}.
    stateOfLight :> {light1 → On, light3 → Off}.
    brightnessLvl :> {light2 → 60}.
}
```

This FO(·) description outlines the functionality of lights within a smart home. The following sections break down and explain each block in detail.

Vocabulary The vocabulary is the first part of the FO(·) description and specifies the domain of the problem. In this case it defines how a light is represented, as well as the state of the light. There is extra info about the possible states of the light, as the value of "State" can only be "On", "Off" or "Dimmed". Additionally, the vocabulary defines a subtype called "SimpleState", which is a subtype of State. A subtype narrows the domain of its parent type by introducing further restrictions. In this example, "SimpleState" includes only "On" and "Off", excluding the "Dimmed" state. This allows the model to distinguish between general light states and a more limited set of "simple" states, while still treating all "SimpleState" values as valid "State" values. There are two functions defined: "stateOfLight" and "brightnessLvl". Both map a "Light" to a "State" or an Integer, respectively. The third function, "isSomeoneHome", does not take any arguments and returns the built-in type "Boolean".

- "stateOfLight": it takes a "Light" and maps it to a "State". In effect, it represents the state ("On" or "Off") of a given light.
- "brightnessLvl": it takes a "Light" and maps it to an Integer. In effect, it represents the brightness-level of a given light.
- "isSomeoneHome": it does not take any arguments and represents a Boolean ("True" or "False"). This is called a proposition. It could be seen as a constant that does not change in one structure.

Theory We will now go over the four formulas in the theory.

```
∀l in Light: ¬isSomeoneHome() ⇒ stateOfLight(l) = Off.
```

The first rule ensures that *if* nobody is home, *then* all the lights should be off.
In the FO(·) world, this is called an implication. If the statement on the left of the implication symbol (⇒) is true, then the right statement also needs to be true. The "∀l in Light" part means that the rule following this statement, can use the variable "l" to refer to any light of type "Light". In other words, it quantifies over all the instances of the type "Light".

```
∀l in Light: (brightnessLvl(l) ≥ 0) ∧
(brightnessLvl(l) ≤ 100) ∧ (brightnessLvl(l) % 10 = 0).
```

This rule ensures that the brightness level of a light remains within the range of 0 to 100. Additionally, it enforces that the brightness increments are in multiples of ten, using a modulo operator.

```
∀l in Light: stateOfLight(l) = Off ⇔
(¬isSomeoneHome() ∨ (brightnessLvl(l) = 0)).
```

The third rule ensures that the light is off if either nobody is home or if the brightness level is 0, and vice versa.
This kind of rule uses an “if and only if” construct, which is more commonly referred to as an equivalence. If the left formula is true, the right one is too, and vice versa. Consequently, if one is false, the other must be too.

```
{
   ∀l in Light: stateOfLight(l) = Off ← brightnessLvl(l) ≤ 0.
   ∀l in Light: stateOfLight(l) = On ← brightnessLvl(l) > 0.
}
```

A light is defined off, when the brightness level is equal to or less than 0. A light is defined on, when the brightness level is greater then 0.
This last rule is a definition. It is a convenient way to define a concept, in this case defining how a light should behave. All the relevant rules are grouped together, which makes it more elegant and more expressive. Important to note is the fact that definitions, unlike implications, need to be fully enumerated and need to capture the necessary and sufficient conditions. This means that all the possible cases need to be defined and when one of the conditions is met, it leads directly to the corresponding output. In this case, the light can be “On” or “Off”, so the definition has defined both of these situations.

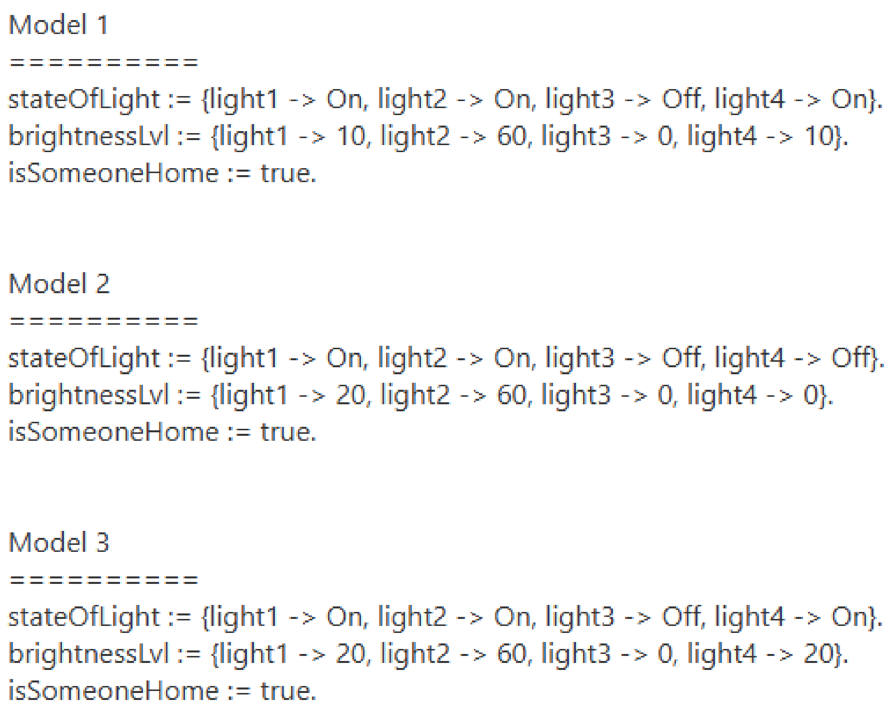
Structure In the structure, a specific situation is defined. In this case, we declare that there are 4 lights and partially map some lights to states and brightness levels. The “:>” symbol stands for a partial interpretation, this means that not all the lights are mapped here.

2.3 IDP-Z3

IDP-Z3 [4] is a reasoning engine capable of performing a variety of reasoning tasks on knowledge bases in the FO(·) language. The idea is to provide knowledge (in the form of FO(·)) that is used by the inference tasks of the IDP-Z3 system to produce an output. Because IDP-Z3 is built to implement the knowledge-base paradigm, it supports multiple inferences.

2.3.1 The Knowledge-Base Paradigm

The IDP-Z3 engine implements the Knowledge Base paradigm [5], in which systems store declarative domain knowledge in a knowledge base and use it to solve a variety of problems. Importantly, it states that the knowledge base should be separated from its inference tasks. This implies that the knowledge could be reused for multiple use cases within the same domain, unlike an imperative programming language where every inference would need its own separate program. Furthermore, if the KB were to change, the programs for the different inference tasks would need to be rewritten to adapt to the new knowledge. This is where



```
Model 1
==========
stateOfLight := {light1 -> On, light2 -> On, light3 -> Off, light4 -> On}.
brightnessLvl := {light1 -> 10, light2 -> 60, light3 -> 0, light4 -> 10}.
isSomeoneHome := true.


Model 2
==========
stateOfLight := {light1 -> On, light2 -> On, light3 -> Off, light4 -> Off}.
brightnessLvl := {light1 -> 20, light2 -> 60, light3 -> 0, light4 -> 0}.
isSomeoneHome := true.


Model 3
==========
stateOfLight := {light1 -> On, light2 -> On, light3 -> Off, light4 -> On}.
brightnessLvl := {light1 -> 20, light2 -> 60, light3 -> 0, light4 -> 20}.
isSomeoneHome := true.
```

Figure 2.3: 10 possible models in line with the provided theory and structure

the power of a knowledge-based system lies.
The multiple inference tasks are explained below using the example of the lights, shown in Section 2.2.

Model expansion In the first place IDP-Z3 can generate models based on a given vocabulary, theory and structure. A model is a complete set of values that satisfies the theory. As an example, Figure 2.3 shows 3 possible models of the lights example.

Propagation The reasoning engine can compute all the logical consequences of a theory. In the first implication of the theory, it is stated that if no one is home, the light needs to be in the "Off" state. So, if the lights are on, IDP-Z3 can infer that there must be someone home.

Explanation IDP-Z3 can provide an explanation to certain models and tell the user why one can or cannot exist. This is a task that, among other uses, is used in the Interactive Consultant UI in the online IDE, which is shown below in Figure 2.4 and is further explained in Section 2.3.2.

Optimisation The reasoning engine can optimize models. For example, the second rule of the theory states that the brightness-level of a light needs to be between 0 and 100 and that it is divisible by 10. It can work out that the smallest possible number of the brightness is 0. An example of this is shown in Figure 2.6

Relevance IDP-Z3 can figure out if there are any irrelevant symbols in the KB. Some rules of the theory are repeated below to further explain this. Rule 1 states that a light is "Off" either when the brightness-level of that lamp is 0 or nobody is home (and vice versa because it is an equivalence). However, not all the symbols this rule produces are meaningfull for the KB:
The definition states that a light is off when the brightness level of said light is 0 or smaller. So when the light is turned off, the constant "isSomeoneHome()" is not relevant anymore since it only defines when the light should be off.

```
{
   ∀l in Light: stateOfLight(l) = Off ← brightnessLvl(l) ≤ 0.
   ∀l in Light: stateOfLight(l) = On ← brightnessLvl(l) > 0.
}
∀l in Light: ¬isSomeoneHome() ⇒ stateOfLight(l) = Off.
∀l in Light: stateOfLight(l) = Off ⇔
(¬isSomeoneHome() ∨ (brightnessLvl(l) = 0)).
```

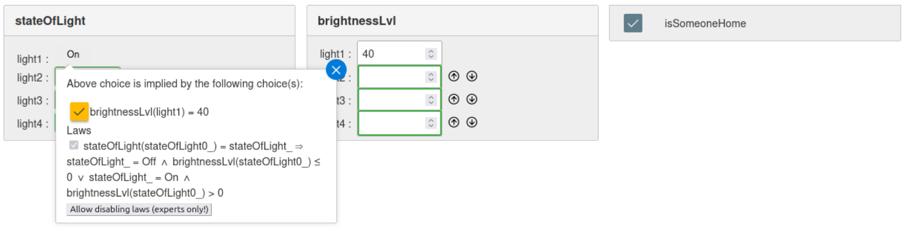
2.3.2 Interactive Consultant

The Interactive Consultant [6] (IC) is a UI integrated into the online IDE of IDP-Z3, allowing users to interact with and test their constructed KB. Each time the user enters a value, the IC automatically adjusts the other values so that they are in line with the KB. This can be a great tool to get insight in a KB. The usefulness is further explained using the example KB from Section 2.2.

As shown in Figure 2.4, the IC can explain to the user why certain behavior is implied. In this case, the KB states that a light is "On" if the brightness level of that light is greater than 0. So when the user sets the brightness level of "light1" to 40, the light should be "On". The "isSomeoneHome()" predicate also became "True". This is because if no one is home, all the lights should be "Off". This is a great showcase of how IDP-Z3 can propagate value assignment.

It can also explain to the user why a combination of values, created by the user, cannot exist, as shown in Figure 2.5. The user wanted to make the brightness level of "light2" 45, which is in conflict with the rule that states that the brightness level of every light must be between 0 and 100 and be divisible by 10.

The IC can also optimize the value of numerical symbols. As shown in Figure 2.6, "light1" is minimized by pressing the button hovered over by "light2". The brightness level is 0, because if the light is "Off", its brightness level can be 0. It can not be any lower because of the rule that states that the brightness level of any light must be between 0 and 100 and be divisible


stateOfLight

light1 : On

light2 :

light3 :

light4 :

Above choice is implied by the following choice(s):

✓ brightnessLvl(light1) = 40

Laws

✓ stateOfLight(stateOfLight0_) = stateOfLight_ ⇒ stateOfLight_ = Off ∧ brightnessLvl(stateOfLight0_) ≤ 0 ∨ stateOfLight_ = On ∧ brightnessLvl(stateOfLight0_) > 0

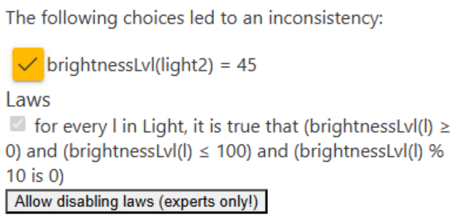
Allow disabling laws (experts only!)

brightnessLvl

light1 : 40

✓ isSomeoneHome

Figure 2.4: An example of how the Interactive Consultant explains a model


The following choices led to an inconsistency:

✓ brightnessLvl(light2) = 45

Laws

✓ for every l in Light, it is true that (brightnessLvl(l) ≥ 0) and (brightnessLvl(l) ≤ 100) and (brightnessLvl(l) % 10 is 0)

Allow disabling laws (experts only!)

Figure 2.5: An example of how the Interactive Consultant explains why a model can not exist

by 10. "light3" however is turned on, so its brightness level cannot be 0. It also cannot be 1 because of the same rule that limits the brightness level.

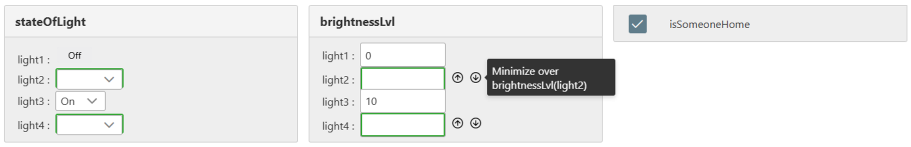
2.4 Alternatives for FO(·)

Given that FO(·) can be difficult to learn and understand for non-expert users [7] [8], it is useful to explore other options for more accessible notations. In this section, we consider alternative options in the context of home automation.

2.4.1 DMN

The Decision Model and Notation (DMN) standard [8] is a user-friendly notation for decision logic. It is managed by the Object Management Group (OMG), and aims to be an intuitive language that can be used by anyone involved in the modeling process. A DMN model has two main components:

- Decision Requirements Diagram (DRD)
- Decision Tables (DT)


stateOfLight

light1 : Off

light2 :

light3 : On

light4 :

brightnessLvl

light1 : 0

light2 :

light3 : 10

light4 :

Minimize over brightnessLvl(light2)

✓ isSomeoneHome

Figure 2.6: An example of how the Interactive Consultant optimizes over something

Define state of light		
U	brightnessLevel	stateOfLight
1	≥ 0	on
2	≤ 0	off

Figure 2.7: An example of a Decision Table using the example of the lights

Decision Tables A DT describes a decision. It defines the output based on a set of input variables. It requires these variables to be fully enumerated, meaning that all possible combinations must be covered. It also needs to capture the necessary and sufficient conditions. These conditions ensure that all criteria must be met to produce a particular output, and when one condition is met, it leads directly to the corresponding output.
The example shown in Figure 2.7 represents the definition of the lights example, of which the FO(·) representation can be found below. It states that *if* the brightness level of a light is greater then 0, *then* that light is "On"; *if* the brightness level of a light is 0, *then* that light is "Off". Important to note is that all DTs can be represented by an FO(·) definition, but not all FO(·) definitions can be represented by a DT. Similarly, implications cannot be represented in a DT. This is a problem, because an implication is a very important part of FO(·) and it is needed for modeling a home.

```
{
   stateOfLight() = Off ← brightnessLvl() ≤ 0.
   stateOfLight() = On ← brightnessLvl() > 0.
}
```

Pros and cons The main advantage of using DMN is its clarity and user-friendliness. The user does not need any programming knowledge to model with it effectively. However, a significant drawback is its limited capabilities. When the user attempts to describe more complex situations, it can quickly become very unreadable, so much so that some solutions to those situations fail to meet DMN's readability goals.

cDMN cDMN [9] is an extension of DMN that addresses the shortcomings of standard DMN by introducing constraint modeling, quantification, types, and functions. By adding these things, an implication is possible, as shown in Figure 2.8. Not all possible states of "isSomeoneHome" are described in the table. If no one is home, the light should be off, it does not specify what should happen if someone is home. The FO(·) code that represents this table is shown below.

```
¬isSomeoneHome() ⇒ stateOfLight() = Off.
```

Disable lights if nobody's home		
E*	isSomeoneHome	stateOfLight
1	No	off

Figure 2.8: An example of a cDMN table using an implication of the example of the lights

2.4.2 CNL

A Controlled Natural Language [10] (CNL) is a subset of a natural language (e.g. English) that is strictly defined. This allows it to be understood by both a computer and a human. This is useful because the domain experts do not need an expert on the language. According to [11], a CNL consists of two parts:

- A subset of a language used as "syntax"
- A parsing engine so the computer can understand the language

IDP-Z3 has its own CNL, which we will use as an example.

The IDP-Z3 CNL The example of the lights, shown below in CNL form, is used to further explain the concept of CNL. This natural language representation is likely to make it more readable to the average person. Another way in which the online IDE makes use of this CNL is in its Interactive Consultant. In the screenshot shown in Figure 2.4, the explanation is useful and helps the user understand what is going on, but it still requires some understanding of FO symbols, which is not ideal. The example below shows the CNL as a replacement for FO(·). While it is more readable, there is still a clear reference to the FO(·) language. It feels unintuitive to say "for all l in Light", instead of "for every Light l". While the second phrasing is clearer for humans, the order change makes it so the IDP-Z3 system can not understand it.

```
{
for all l in Light: stateOfLight(l) = Off if brightnessLvl(l) ≤ 0.
for all l in Light: stateOfLight(l) = On if brightnessLvl(l) > 0.
}
```

Pros and cons A CNL allows the user to create easy-to-read code because it uses a subset of a known language, removing the need of a technical expert on the used language. However, it is not the easiest to write due to its strictness with the ordering of words, which can introduce confusion. Two structures that seem the same to us humans, may be interpreted differently by the computer. The limited set of words available in a CNL can also pose challenges. While the user might prefer to write in natural, unrestricted text as they are used to, the computer is unable to interpret such input. This challenge, known as the writability problem [10], highlights the difficulty of creating a language that remains readable without introducing ambiguity, so the computer can still understand it. These restrictions can lead to an unintended consequence: writing in a CNL may become more challenging than learning the actual language it is meant to simplify. Users can become frustrated because they need to learn the boundries of the CNL as well as the syntax.

2.4.3 Blocks-based editor

A blocks-based editor is a user-friendly way to write software using blocks. This allows the user to avoid memorizing syntax, as the blocks are provided and visually indicate what is possible and what is not, eliminating the need to understand complex syntax rules. In other words, syntax errors are non-existent, allowing the user to fully focus on what the application should do, rather than how to write it in a certain language.

A well-known example of a blocks-based programming enviroment is Scratch [12], a blocks-based interface in JavaScript. It allows users to create programs in a visual workspace, while processing the blocks into runnable JavaScript code. Scratch is a code editor designed for children, ensuring that it needs to be understandable even for the youngest users.

FO(·) blocks-based editor There already exists a blocks-based editor for FO(·) [13]. It uses Blockly [14], an extendable blocks-based editor developed by Google. It has two components:

- A workspace where blocks can be placed
- A generator where blocks are translated to a textual representation (in this case FO(·))

An example of this editor is shown in Figure 2.9. In this editor the three main blocks of the FO(·) language are still clearly visible as well as the form of the FO(·)-syntax. The vocabulary has types, functions and predicates to define the problem domain. The theory has a formula and a definition, where the underlying FO(·)-syntax is still very apparent. The first block of the vocabulary can be easily translated into FO(·)-syntax, as shown below. As this is a blocks based editor for the entire FO(·) syntax, it is likely too complex for what is needed for IoT in home automation. Still, it could provide a good baseline to start from.

```
type Vertex := {1..3}
```

Pros and cons A blocks-based editor is very clear and easy in how to use it. The user can not make syntax errors, because they can see what blocks fit each other and which do not. This implies that no expert of the language (or blocks) is needed to create a working project. However, by making it all visual, it can get messy quite quickly. For bigger projects, that need more complex functionality, the workspace can get very unorganised and difficult

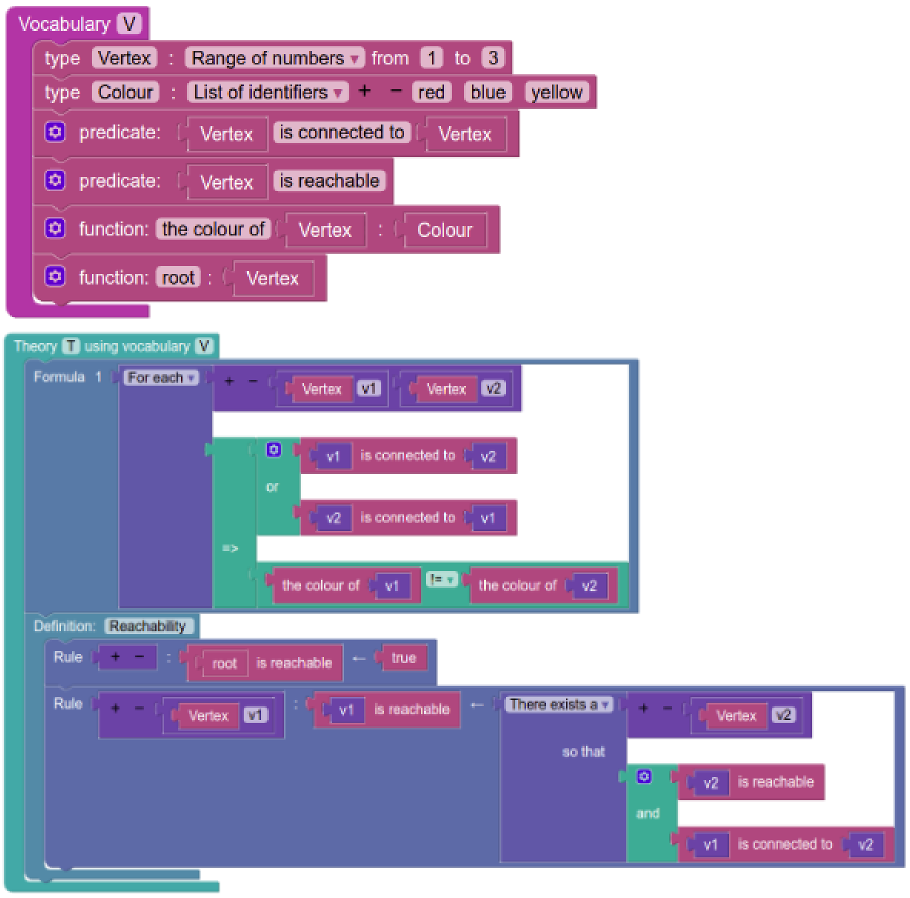

Vocabulary V
type Vertex : Range of numbers from 1 to 3
type Colour : List of identifiers + − red blue yellow
predicate: Vertex is connected to Vertex
predicate: Vertex is reachable
function: the colour of Vertex : Colour
function: root : Vertex
Theory T using vocabulary V
Formula 1 For each + − Vertex v1 Vertex v2
v1 is connected to v2
or
v2 is connected to v1
=>
the colour of v1 != the colour of v2
Definition: Reachability
Rule + − : root is reachable ← true
Rule + − Vertex v1 : v1 is reachable ← There exists a + − Vertex v2
so that
v2 is reachable
and
v1 is connected to v2

Figure 2.9: An example of a Blockly editor for FO(·) [13]

Table 2.2 Comparison of possible user-friendly interfaces FO(·)

User-friendly interface	Graphical interface	Low code	Sufficiently expressive
DMN	x	x	
IDP-Z3 CNL			x
Blocks-based editor	x	x	x

to navigate. There are some ways to try and fix this, for example by creating a new block with combined funtionality which is described in its own workspace.

An overview of the meaningful pros and cons of all the discussed user-friendly interfaces can be found in Table 2.2.

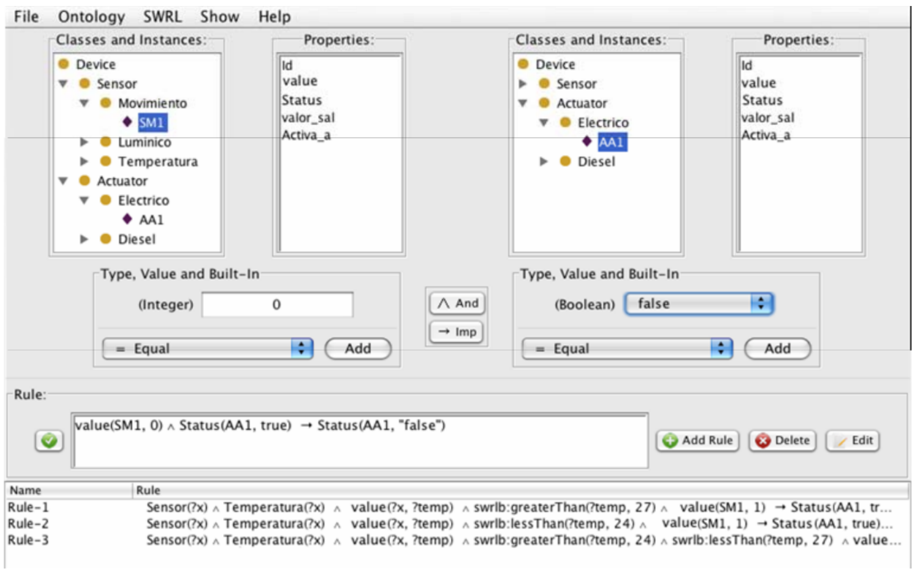
2.5 State of the art

When considering how AI can help both with user-friendliness and how the user describes their home automation system, according to [15], knowledge-based AI fits into this perfectly. It is easier for non-experts to create a knowledge-based home automation system than it is to create a traditional imperatively programmed system because a KB system is much more flexible. However, it still requires the user to modify the rules of the KB when circumstances change.

2.5.1 IntelliDomo

IntelliDomo [16] is an ontology-based interface for home automation. It uses an ontology (OntoDomo) to represent the KB, production rules written in Semantic Web Rule Language (SWRL) to describe the system, and an inference engine (Jess) to make decisions.

The KB is represented in OWL files (Ontology Web Language), which are created by the system using existing ontologies. This is convenient when an appropriate ontology already exists; however, if there is none, the user must understand OWL to create their own. IntelliDomo uses SWRL in combination with a custom-made UI for rule construction. This ensures that users cannot make syntax errors, as all possible options are predefined. However, the UI still has the SWRL syntax in it, so it still requires the user to know SWRL even tough they do not need to write any. It can be challenging for non expert users to follow the SWRL logic. As shown in Figure 2.10, the rules are constructed in the lower part of the UI using the predefined upper part. The constructed rules are difficult to read and understand once created. If there were to be a mistake made by the user, they would be required to change the rules in the SWRL syntax or completely reconstruct the rule from scratch. It also fails to give the user any feedback on the functionallity of thhere created rules, which is needed as explained in Section 2.1.3.



Rule:

value(SM1, 0) ∧ Status(AA1, true) → Status(AA1, "false")

Name	Rule
Rule-1	Sensor(?x) ∧ Temperatura(?x) ∧ value(?x, ?temp) ∧ swrlb:greaterThan(?temp, 27) ∧ value(SM1, 1) → Status(AA1, tr...
Rule-2	Sensor(?x) ∧ Temperatura(?x) ∧ value(?x, ?temp) ∧ swrlb:lessThan(?temp, 24) ∧ value(SM1, 1) → Status (AA1, true)...
Rule-3	Sensor(?x) ∧ Temperatura(?x) ∧ value(?x, ?temp) ∧ swrlb:greaterThan(?temp, 24) ∧ swrlb:lessThan(?temp, 27) ∧ value...

Figure 2.10: The IntelliDomo UI [16]

2.5.2 User-configurable semantic home automation

User-configurable semantic home automation [3] (USHAS) is another system that uses a knowledge-based approach to home automation. They use the Web Ontology Language [17], OWL for short, as a language for their ontology. They have six main concepts, also called first-level classes:

- Person, to represent people present in the home. The class is split into subclasses like child, adult... Each person has two properties: a name and their location.
- Location, to represent the different rooms in the house. These are connected semantically, which means that the system can understand the relationships between them. For example, a kitchen is part of the first floor, which is part of the house. This enables the user to create rules that can apply to all devices located in a certain room.
- Device, to represent the devices in the house. The class is split into two subclasses: "ApplianceCategory" (devices with the same states, for example a light and a lightswitch) and "FunctionalCategory" (devices with the same function, for example a remote volume controller and the manual knob on the box). Each device has a location, states, and events.
- Time, to represent the time of day. This is important for home automation, as it can be used to trigger certain actions at specific times. It is also capable of representing repeating events, like "every Monday at 14:00".
- Environment, to represent the environment in and around the house. This includes things like temperature, humidity, light, and noise levels, which can be used to trigger

certain actions. Also, user-defined variables can be expressed by creating an extra environment property.

- Event, to represent the events that can occur in the house. This includes things like a person entering or leaving a room, a device being turned on or off, or a change in the environment. These events can be used to trigger certain actions.

The USHAS system uses a CNL-like UI to allow users to create their own ontology and rules, which are written in their custom Semantic Home Process Language (SHPL) in XML files. These consist of four main components: preconditions, variables, execution time, and flow of invocations. For example, given a command "if the temperature is higher than 30 °C at 6:00 PM, turn on the air conditioner", the temperature is a variable; "if the temperature is higher than 30 °C" is a precondition, 6:00 PM is an execution time, and "turn on the air conditioner" is an invocation. All of these can be found in their UI. However, this way of creating rules, can be confusing for users who are not home in this field. As Figure 2.11 shows, one rule requires many different components that might not be intuitive for the user to fill in.

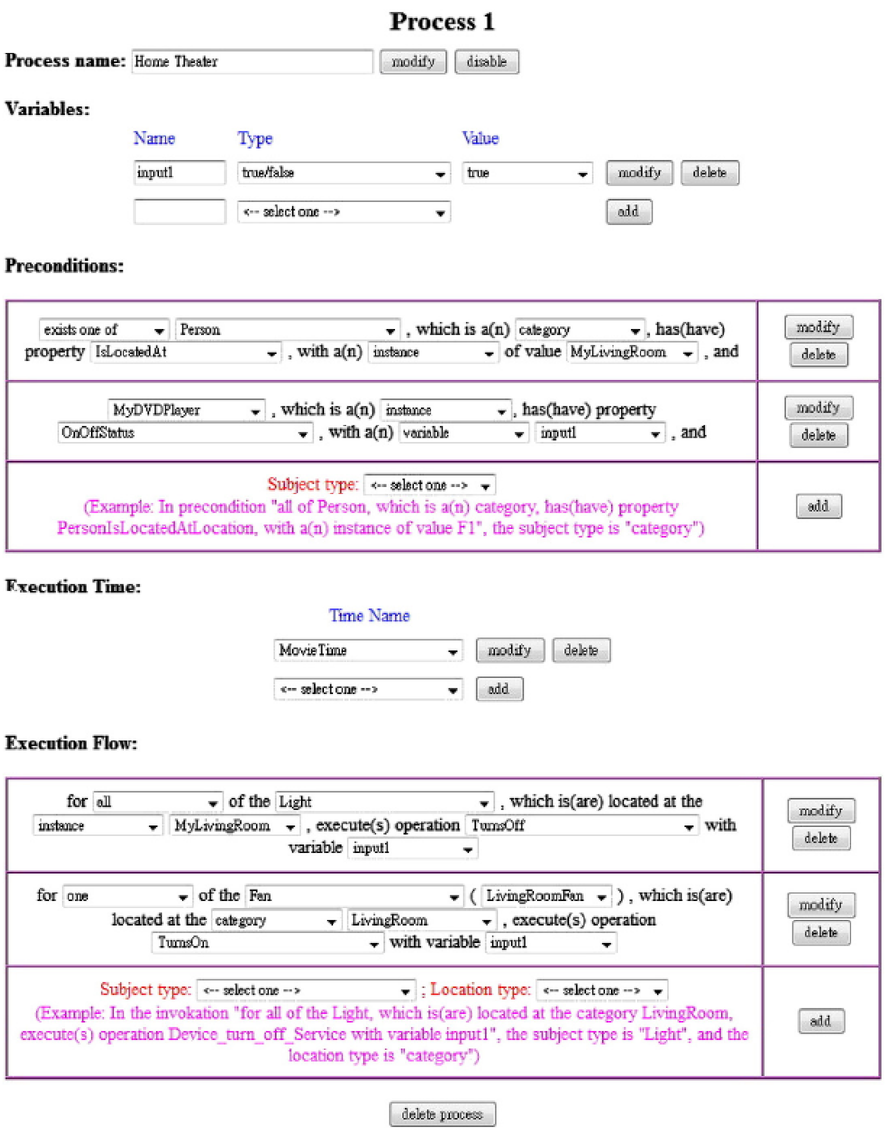
These papers demonstrate that there is a value in a knowledge-based approach to home automation, but it is important to note that their approach to model homes and create rules is not the most user-friendly one. This is however very important when it comes to home automation for laypeople.

2.5.3 Smart Block

SmartBlock [18] is a user-friendly blocks-based interface for the SmartThings IoT platform. They built their editor using Blockly [14], a JavaScript framework for creating blocks-based editors. An example of what the editor looks like is shown in Figure 2.12. It describes that if "lock1" has gone from "locked" to "unlocked" and "presenceSensor1" has detected something, then "light1" should turn on and "musicPlayer1" should start playing. The blocks are colour-coded to make it easier for users to understand what blocks should go together. It is also worth mentioning that they added tools to check for inconsistencies and mistakes made by the user.

They conducted a user study with 33 participants from diverse backgrounds (including IT). First, the participants were asked to model a rule to get familiar with the editor. 81% of them responded positively or neutrally to the ease of use. After this introduction, the participants were tasked with modeling three progressively more challenging rules. Over time, they got better at this, which was shown by how much faster they were able to complete the tasks. Interestingly, the difference in performance between non-IT and IT participants was not that big, which further shows how user-friendly the editor is.

Another test required the participants to identify some errors in a given SmartApp. This proved to be more difficult without assistance, as the debugging tools lacked any semantic



Process 1

Process name: Home Theater [modify] [disable]

Variables:

Name	Type	Value	
input1	true/false	true	[modify] [delete]
	<-- select one -->		[add]

Preconditions:

exists one of Person, which is a(n) category, has(have) property IsLocatedAt, with a(n) instance of value MyLivingRoom, and	[modify] [delete]
MyDVDPlayer, which is a(n) instance, has(have) property OnOffStatus, with a(n) variable input1, and	[modify] [delete]
Subject type: <-- select one --> (Example: In precondition "all of Person, which is a(n) category, has(have) property PersonIsLocatedAtLocation, with a(n) instance of value F1", the subject type is "category")	[add]

Execution Time:

Time Name

MovieTime [modify] [delete]

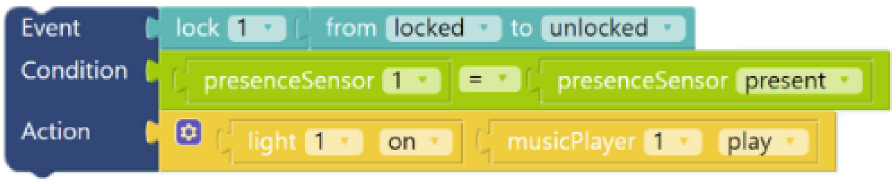
<-- select one --> [add]

Execution Flow:

for all of the Light, which is(are) located at the instance MyLivingRoom, execute(s) operation TurnsOff with variable input1	[modify] [delete]
for one of the Fan (LivingRoomFan), which is(are) located at the category LivingRoom, execute(s) operation TurnsOn with variable input1	[modify] [delete]
Subject type: <-- select one -->; Location type: <-- select one --> (Example: In the invokation "for all of the Light, which is(are) located at the category LivingRoom, execute(s) operation Device_turn_off_Service with variable input1", the subject type is "Light", and the location type is "category")	[add]

[delete process]

Figure 2.11: The USHAS UI [3]



Event: lock 1 from locked to unlocked

Condition: presenceSensor 1 = presenceSensor present

Action: light 1 on; musicPlayer 1 play

Figure 2.12: The SmartBlock UI [18]

error correction. This highlights why having a tool to correct mistakes during the modeling process is so important.

This paper proves that a blocks-based editor is a viable option for home automation. It shows that such an editor is user-friendly and accessible to people without an IT background. It also confirms the importance of tools that can help with semantic errors.

2.5.4 Event based home automation

Another approach to home automation is an event-based system [19]. Instead of thinking in terms of states and their changes, it focuses on events and the logical consequences of those events. For example, if the front door opens, the lights in the hallway should turn on. After that, it is plausible that the person who just walked in will either walk to the living room, kitchen, or upstairs, so the system can predict this and narrow down the possible next actions that need to be taken. So instead of modeling different rules, this approach focuses on a stream of events that lead to a specific goal and making that as user-friendly as possible.

The formalism used to achieve this is called Event Calculus. It is used for representing and reasoning about events and their effects over time. It is a first-order logic system and therefore allows for the representation of complex temporal relationships between events. It is also possible to reason over the written rules. An example of how this could be done is explained with the process of preparing a cup of tea. Normally, one would boil water, grab a cup and a teabag, after which the water is poured into the cup and the teabag is added. The initial action (turning on the kettle) is the event that starts the process, so from that point, the system can predict what the next steps should be. If one of the steps is not done, it can be inferred and explained to the user. In that way, the system can help users who are unfamiliar with the process or users who tend to forget the necessary steps. Another useful side effect of using a first-order-based system is the fact that it can be updated as new data comes in. This means that the system can learn from the actions of the user and adapt to their preferences. For example, if a user always turns on the kettle when they get home, the system can learn this and automatically turn on the kettle when it detects that the user gets home. The initial action is now the user arriving home.

However, all this knowledge needs to be provided by the user, which is not done in the most user-friendly way. They use a first-order logic language to describe these event flows, which, for a non-expert, is not easy to write nor understand. The tea example, in the formalized language, is shown in Figure 2.13. However, with the learning capabilities of the system, this is somewhat masked, as the system can learn from the user and adapt to their preferences.

$Happen(takeTo(kettle, basin), t_0);$

$Happen(turnOn(tap), t_1);$

$Happen(add(cold\ water, kettle), t_2, t_3);$

$Happen(turnOff(tap), t_4);$

$(Happen(boilWater, t_5, t_6)) \vee (Happen(takeTo(mug, kitchen\ table), t_5));$

$Happen(takeTo(teabag, kitchentable), t_{51});$

$Happen(takeTo(milk, kitchen\ table), t_{52});$

$Happen(takeTo(sugar, kitchen\ table), t_{53});$

$waterBoiled\ ?\ Happen(add(teabag, mug), t_7);$

$(Happen(add(boiledWater, mug), t_8));$

$Happen(add(sugar, mug), t_9);$

$Happen(add(milk, mug), t_{10});$

$Happen(stirAround, t_{11}, t_{12});$

$Happen(takeTo(milk, fridge), t_{13});$

$Happen(takeTo(sugar, cupboard), t_{14})$

$Here\ t_0 < t_1 < t_2 < t_3 < t_4 < t_5 < t_6, t_5 < t_{51} < t_{52} < t_{53} < t_6, \wedge t_7 < t_8 < t_9 < t_{10} < t_{11} < t_{12} < t_{13} < t_{14}.$

Figure 2.13: An example of the Event Calculus code for making a cup of tea [19]

Chapter 3

An FO(·) description for home automation

In this chapter, we will discuss different ways to model a home automation system in FO(·). However, before we can do that, we need to figure out what the user expects from their home automation system, and how we can achieve a meaningful and logical description that offers a lot of flexibility while keeping it straightforward enough to be recreated using "simple" blocks. We will discuss the advantages and disadvantages of these implementations, and which of these we will use going forward.

3.1 Home automation description

To decide on the modeling approach for the home automation system, we use a goal-oriented approach. This means that we will look at what the system needs to do and how the user will want to interact with it, and design our system accordingly. We will discuss some example situations that people would want to model. By creating situations, we also have a good idea of what the limitations of our FO(·) description can be.

Devices Devices are the cornerstone of any smart home, so we will need to model these as intuitive as possible, making it so there is no need to understand how the devices are configured in the application. Each device has a number of possible states it can be in. For example, a lightswitch can be "On" or "Off", or a blind can possibly be "opened" or "closed". The user could configure these states themselves, but most devices already have a standard set of states.

Areas The user will want to configure their devices in a way that matches the layout of their home so that they can create rules that apply to devices in a room, which can be very handy. The user can, for example, create a rule that turns off all the lights in the bedroom

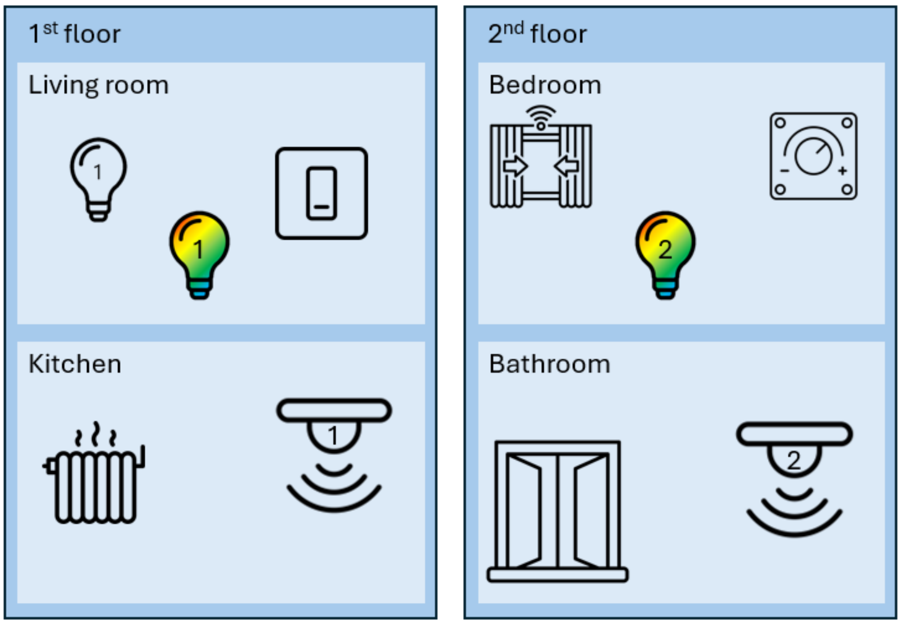


Figure 3.1: An example of a home automation system used to explain the different FO(·) implementations

when nobody is in said bedroom. Since they will set up the areas to match their house, it will create a new way for them to understand what they are doing and therefore make the system more intuitive to use. It is similar to USHAS' [3] approach, where all the areas are semantically connected, as explained in Section 2.5.2. This also enables the user to create rules for "super-areas", which must also hold for all the areas that fall under it. For example, all the areas are part of the house, the user can create a rule that turns off all the lights when they leave the house.

Behaviour It is important to think about what the user would want to configure and what rules they would want to create. Below are some examples of such rules, in increasing complexity.

R1. Turn on a light when a lightswitch is turned on.

R2. Set the power of a fan to 50% when the temperature sensor gives a value of 25°C.

R3. Set an RGB light to yellow when the lightswitch is turned on.

R4. Open a blind and trigger an alarm when it is 8 o'clock.

R5. Turn on all the lights in the house when motion sensor 1 or 2 is triggered at 2 o'clock.

R6. Close all the blinds in the house and set the thermostat to 16°C when it is 10 o'clock.

R7. Turn off all the lights in the living room and turn on all the lights in the kitchen when motion sensor 1 is triggered.

3.2 FO(·) knowledge base

The vocabulary is an important part of the FO(·) specification as it defines the possibilities and limitations for the rest of the specification. There are lots of different approaches possible, which we will explore with a focus on ease of use and effectiveness in a real-world application. We will use the rules described in Section 3.1 as examples of what the user would want to be able to model. The theory, and therefore the possible rules, is directly affected by the choice of vocabulary, so this also needs to be considered.

Approach 1 As a first approach, consider a vocabulary where all the devices are defined as constants, and every device is linked to an area as shown below.

```
vocabulary V {
    type Area := {living_room, kitchen, floor_1, floor_2, house}

    light1: → Area
    type stateOfLight := {0..100}
    stateOfLight1: → stateOfLight

    lightSwitch: → Area
    type stateOfLight_switch := {on, off}
    stateOfLight_switch: → stateOfLight_switch
    .
    .
    .
}

theory T:V {
    stateOfLight_switch() = off ⇒ stateOfLight1() = 0.
}
```

The problem with this way of representing the devices is that there is no way to group them, which is required, for e.g., R7. The user also cannot implement any form of grouping in the devices; take R5 as an example, it is not possible to enumerate over all the light devices using this vocabulary. The states of these devices are represented through a type that defines all possible states for that device. In combination with this, there is a function that represents what state the device is in.

Another shortcoming of this vocabulary is the inability to represent relations between areas. This is because the vocabulary does not allow grouping of devices or areas. This is also something the user would want otherwise R6 would be very tedious to model.

Approach 2 As a second approach, we can add an explicit type "Device" under which all specific instances of devices fall. This allows for the creation of rules that apply to all devices, specific devices, and areas. Rules R1-R7 should all be possible with this vocabulary. It is also capable of representing multiple devices of the same type, e.g., light1 and light2, since these have the same states, they can use the same type for the states. However, the function that describes the state of a device, maps all the devices to all the states, so that, for instance, even light1, light2, etc. have a bell state, which they obviously should not have.

```
vocabulary V {
   type Device := {bell, light1, light2, motionSensor, lightSwitch}
   type Area := {living_room, kitchen, floor_1, floor_2, house}

   deviceIsInArea: Device → Area

   deviceIsBell: Device →  𝔹
   type BellState := {bell_ringing, bell_off}
   bellState : Device → BellState

   deviceIsLight: Device →  𝔹
   type LightState := {light_on, light_off}
   lightState : Device → LightState
}
```

Approach 3 The third approach has a completely different view to the problem. Instead of focusing on the devices themselves, the focus now lies on the states of the devices and their grouping. By using subtypes, different instances of devices can be created according to their state. For example, a motion sensor is binary (motion detected or no motion detected), but a light switch is also binary (on or off), so these can be grouped together. This vocabulary can behave just like the previous one, in the sense that if all devices have different states, it will result in the same behaviour, but it opens up possibilities for more complex rules. By using subtypes, we still only use one function to state what state a device is in, as well as what area it is in. This makes the implementation behind the scenes much easier.

```
vocabulary V {
    type Device := {light1, light2, motionSensor,
    lightSwitch, curtain}
    type State := {on, off, opened, closed}
    type Area := {house, floor1, floor2, kitchen, living_room}

    deviceIsInArea: Device → Area
    areaIsSubAreaOf: Area × Area →  𝔹
    deviceIsInState: Device → State


    type BinaryDevice := {light1, light2,
                          motionSensor, lightSwitch} <: Device
    type BinaryState := {on, off} <: State
}
```

It is also important to note that not all devices use strings as states. For example, a dimmable light, which needs to be set to a specific brightnesslevel, or in practice a number between 0 and 100. This is a problem with the third vocabulary, because we have one function that describes the state of a device, "deviceIsInState", which maps a "Device" to a "State". Because of the fact that "State" only has elements of the type "string" (which can not combine with integers), we can not integrate devices that require integers as states. However, it is solvable by creating a second function that maps a "Device" to an "Int", below called "numberDeviceIsInState". By doing this, we have created a separation in the devices based on how their state requires them to be defined. In our blocks-based interface we should be able to hide this from the user.

```
Vocabulary V {
    type StringDevice := {light_1, light_2}
    type NumberDevice := {dimmable_light}
    type StringState := {light_on, light_off}
    type Area := {home, kitchen, living_room}

    stringDeviceIsInArea: StringDevice → Area
    numberDeviceIsInArea: NumberDevice → Area
    areaIsSubAreaOf: Area × Area →  𝔹

    stringDeviceIsInState: StringDevice → StringState
    numberDeviceIsInState: NumberDevice → Int

    type lightDevice := {light_1, light_2, light_3} <: StringDevice
    type lightDeviceStates := {light_on, light_off} <: StringState

    type dimmableLightDevice := {dimmable_light} <: NumberDevice
    type dimmableLightDeviceStates := {0..100} <: Int
}
```

Having experimented with all three of the possibilities, we have decided to go with the third vocabulary with the split devices for number states and string states. It offers the user the flexibility they want, while keeping it as simple and convenient enough to model in a blocks based editor. The first vocabulary is a bit too simple and does not offer the user the flexibility they want in rule creation, though it is very easy to create blocks for it. The second vocabulary is complexer and offers a great flexibility in the rules that can be created. However, it is quite tedious to always specify the possible states for every device. While the third vocabulary is a bit more complex, we feel it is still easy to understand while making the device-creation easier.

Chapter 4

Implementation

In this chapter we will discuss the blocks-based editor and the design-decisions we have made while creating it. We will also discuss how the application is structured and how it works.

4.1 Blockly

To create the blocks-based interface we used Blockly [14], a JavaScript-based framework for blocks-based editors. It is, at its core, a very flexible framework that allows you to create custom blocks that translate to code-snippets. We used it to create JSON-like structures that can be parsed into FO(·) later on.

Blockly comes with basic programming blocks by default, but since those are not suitable for our FO(·) descriptions, we created a custom set of blocks known as a toolbox. The toolbox defines how blocks are grouped and displayed in the editor. The actual appearance of each block is defined separately, and their behavior, how each block translates into a FO(·)-compatible format, is handled by custom generator functions. Whenever the Blockly workspace updates, a main script ensures that the generated code stays in sync and that elements like dropdown menus reflect the latest state of the workspace.

When creating the custom blocks and deciding where to place them in the editor, we did a few things to make it as user-friendly as possible. We made sure that the blocks that are related to each other, for example the blocks that define an area and the blocks that displays the relations between areas, are close and have the same tint of colour. This is an extra way of showing the user which blocks should be puzzled together. We also tried to make the blocks in a way that hides the underlying FO(·) syntax as much as possible. We also dynamicially update the dropdown menus inside the blocks so the user can, for example, not select a devicetype when a state is needed. In Figure 4.1 all the custom blocks we used are shown. The blocks will be discussed in more detail in the following Section 4.2.

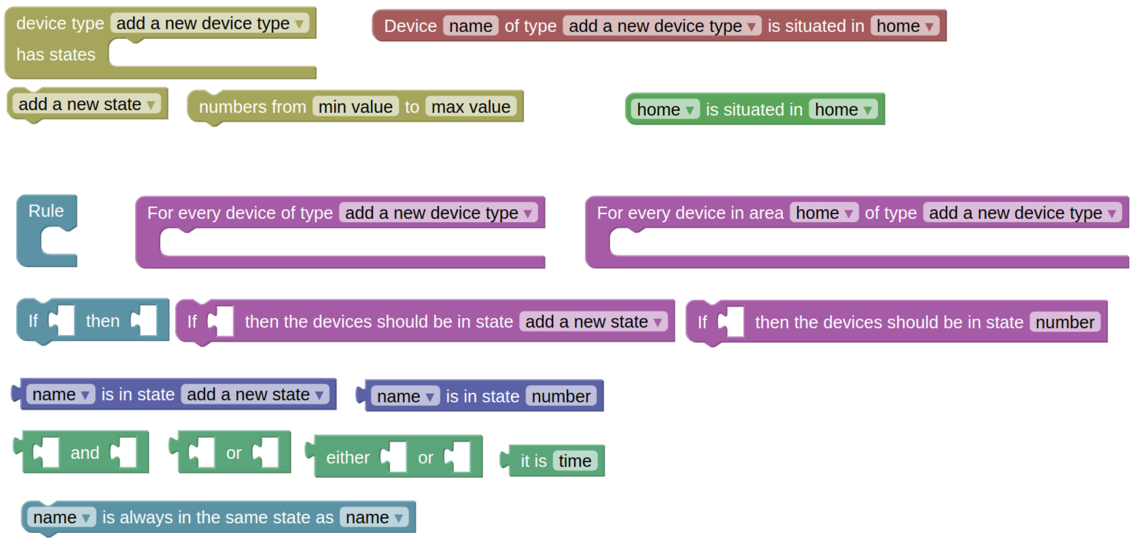

device type add a new device type
has states
Device name of type add a new device type is situated in home
add a new state
numbers from min value to max value
home is situated in home
Rule
For every device of type add a new device type
For every device in area home of type add a new device type
If then
If then the devices should be in state add a new state
If then the devices should be in state number
name is in state add a new state
name is in state number
and
or
either or
it is time
name is always in the same state as name

Figure 4.1: All the custom blocks for the blocks-based editor

4.2 Custom blocks

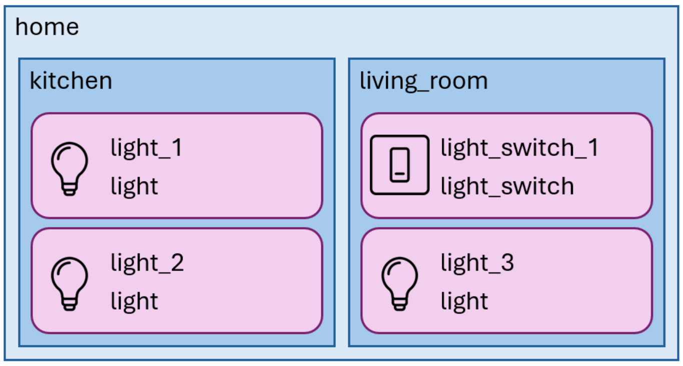
In this section we will go over all the blocks we created, how they translate in FO(·), and how we imagine them being used. In addition to the blocks there are also buttons, e.g., to create a new devicetype or a new area. These will also be discussed here. The focus lies on the blocks and how they translate to FO(·), not on the complete FO(·) description. Throughout this section we will use the example shown in Figure 4.2. Figure 4.2a shows the structure of the home. The rules that go with this home are:

- If "light_switch_1" is in state "light_switch_on", "light_3" should be in state "light_on".
- If "light_switch_1" is in state "light_switch_on", "light_1" and "light_2" should be in state "light_off".
- If "light_switch_1" is in state "light_switch_off", "light_1" and "light_2" should be in state "light_on".
- If "light_switch_1" is in state "light_switch_off", "light_3" should be in state "light_off".

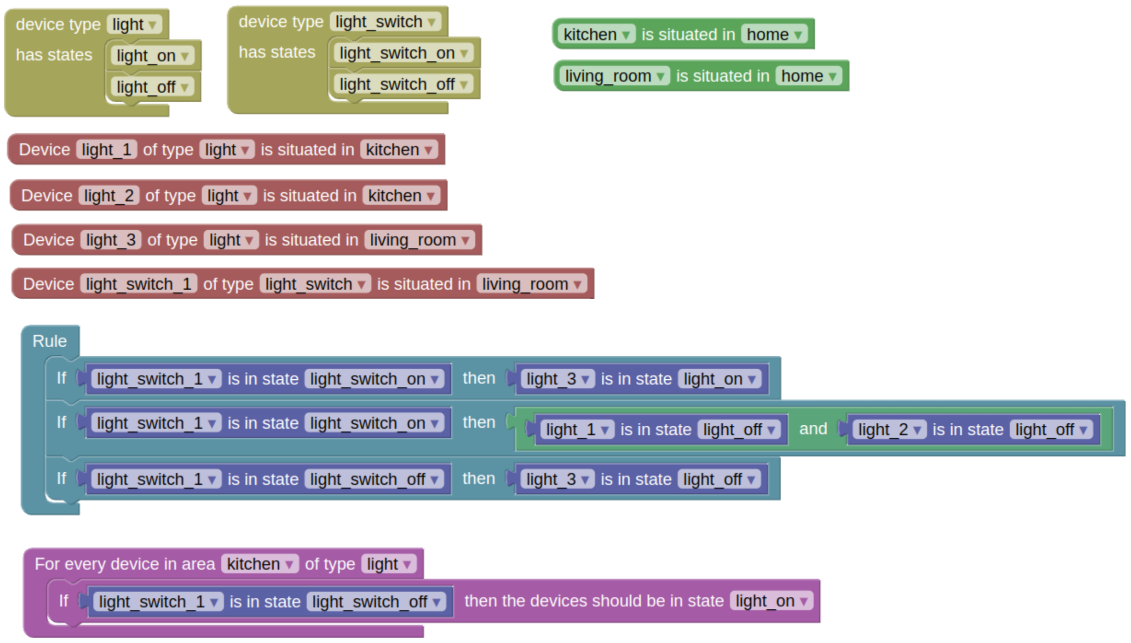
The example modeled in the editor, is shown in Figure 4.2b. We will go over the blocks used in the example to explain how they work.

4.2.1 States

States are used to specify the possible states of a device. For instance, a light can be in the states "light_on" and "light_off". States are created using a button. When the button is clicked, the user gets a prompt asking for the name of the state. After they have entered



(a) An example of a simple home



```
device type light
has states light_on
           light_off

device type light_switch
has states light_switch_on
           light_switch_off

kitchen is situated in home
living_room is situated in home

Device light_1 of type light is situated in kitchen
Device light_2 of type light is situated in kitchen
Device light_3 of type light is situated in living_room
Device light_switch_1 of type light_switch is situated in living_room

Rule
  If light_switch_1 is in state light_switch_on then light_3 is in state light_on
  If light_switch_1 is in state light_switch_on then light_1 is in state light_off and light_2 is in state light_off
  If light_switch_1 is in state light_switch_off then light_3 is in state light_off

For every device in area kitchen of type light
  If light_switch_1 is in state light_switch_off then the devices should be in state light_on
```

(b) The example modeled in the blocks-based editor

Figure 4.2: An example of a simple home and the corresponding blocks-based implementation

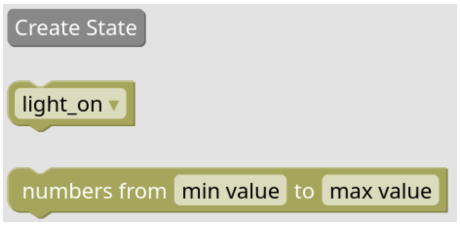


Figure 4.3: The blocks related to the states

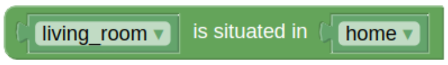


Figure 4.4: An example of defining the relation between two areas

a name, the state is added to the globally defined array that holds all the states. This is done so the blocks that require the states can only display the existing states, which makes it easier for the user to select the right state. This is how states are created for devices that require strings as state. However, some devices require a number as state, for example a thermostat. It should be able to represent a temperature range, e.g. 16-25°C. The difference between the "number-device" and "string-device" is further explained when discussing the devicetypes. There is no immediate translation to FO(·) code when a state is created. All the blocks related to the states are shown in Figure 4.3, and will be further explained in the context with other blocks, since any state-block should not be used on its own.

4.2.2 Areas

An area is a physical location in the house, such as, a living room or a kitchen. Areas are created in the same way as states, using a button that writes new areas to a globally defined array. There is however an extra block that is used to define the relations between the areas. This is so the user can define relations between the areas, for example "the living room is inside the home". An example of this combination can be found in Figure 4.4.

The block has a direct FO(·) translation in the structure as shown below.

```
structure S:V {
    areaIsSubAreaOf := {(living_room, home)}.
}
```

4.2.3 devicetypes creation

A devicetype is a group of devices who share the same states. For example, the devicetype "light" groups all the lights, who have the states "light_on" and "light_off", together. The devicetypes are created in the same way as areas and states. Since the devicetypes still require the user to specify the states of the devicetype, there is no explicit FO(·) translation.

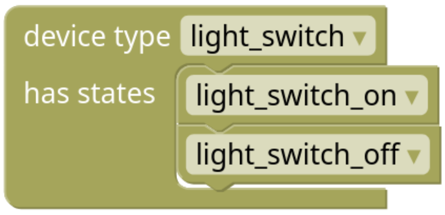


Figure 4.5: An example of defining the states of a device

4.2.4 Devicetypes definition

The yellow block shown in Figure 4.5 is used to specify the states of a given devicetype. There are two different types of devicetypes: "stringDevice" and "numberDevice". However, this does not reflect in the devicetype block since this distinction is already done in the states. Behind the scenes, the program looks whether or not the states that define the devicetype are convertable to numbers, after which it adds the new devicetype to the correct supertype. In the case of a light switch, the devicetype "light_switch" has two states: "light_switch_on" and "light_switch_off". The block is used to define the states of a device, so the user can select the states of a device when creating it. The block has a direct FO(·) translation, both in the vocabulary and theory as shown below. Note that the first line in the vocabulary does not yet specify specific devices that are of the type "light_switch", as these will be added when the user creates a device of that type.

```
vocabulary V {
    type light_switchDevice := {} <: StringDevice
    type light_switchDeviceStates := {light_switch_on,
    light_switch_off} <: StringState
}
theory T:V {
    ∀dt in light_switchDevice: ∃x in light_switchDeviceStates:
   stringDeviceIsInState(dt) = x.
}
```

Note that we also introduce a formula in the theory to state that the device is in one of the states of the devicetype. This prevents, e.g., that a "light_switch" can be set to "light_on".

4.2.5 Devices

A device is created using the red block shown in Figure 4.6. The user needs to specify the name of the device, the devicetype and the area it is in. For example, a device called "light_switch_1" is of the type "light_switchDevice" and is located in the "living_room". The block has a direct FO(·) translation, shown below. However, this again depends on whether it is a number or string device. The line created by the devicetype-block is changed to include the newly defined device.

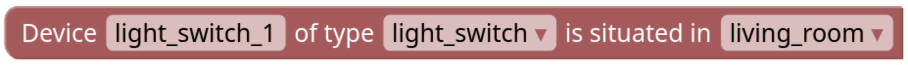


Figure 4.6: An example of a new device

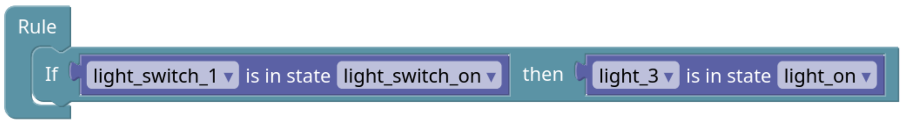


Figure 4.7: An example of a single rule

```
vocabulary V {
    type stringDevice := {light_switch_1}
    type light_switchDevice := {light_switch_1} <: stringDevice
}
Structure S:V {
    stringDeviceIsInArea := {light_switch_1→living_room}.
}
```

4.2.6 Specific rules

The blocks shown in Figure 4.7 are used to create a standalone rule. The user can create a pre- and postcondition as complex as needed. After which the user selects a device of which the state is changed when the first part of the rule is true. For example, if the "light_switch_1" is in the state "light_switch_on", "light_3" should be in the state "light_on". This corresponds to the following FO(·).

```
stringDeviceIsInState(light_switch_1) = light_switch_on ⇒
stringDeviceIsInState(light_3) = light_on.
```

4.2.7 Enumerating rules over devicetypes

The blocks shown in Figure 4.8 are used to create a rule that is true for all devices of a certain type. It only enumerates over one type at once, not multiple at the same time. This is done so there is no confusion in the blocks that need to specify the rule that applies for the devices. If multiple devicetypes were possible, each devicetype should have its own name in the scope of the block to be able to use it, which would make it confusing. The user selects a device which state dictates whether or not the rule is true. After which the user selects a state in which all the devices of that type should be in. For example, if the "light_switch_1" is in the state "light_switch_off", all the lights should be in the state "light_off". The block has a direct FO(·) translation, shown below.

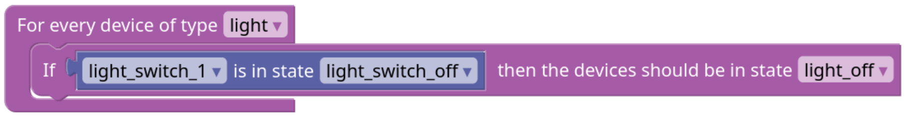

For every device of type light
If light_switch_1 is in state light_switch_off then the devices should be in state light_off

Figure 4.8: An example of the blockly block that creates an enumerating rule

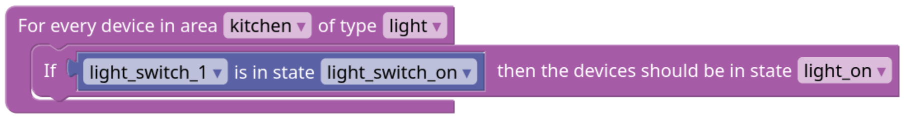

For every device in area kitchen of type light
If light_switch_1 is in state light_switch_on then the devices should be in state light_on

Figure 4.9: An example of the blockly block that creates a quantifying rule over areas

```
∀DT in LightDevice:
stringDeviceIsInState(light_switch_1) = light_switch_off ⇒
stringDeviceIsInState(DT) = light_off.
```

4.2.8 Quantifying rules over areas

The blocks shown in Figure 4.9 are used to create a rule that is true for all devices in a certain area. The user selects an area who limits the selected devices to the devices in that area. The user can now input one of the previous rules to create a rule that applies only in a certain area. For example, if the "light_switch_1" is in the state "light_switch_on", all the lights in the kitchen should be in the state "light_on". The block has a direct FO(·) translation, shown below.

```
∀DT in LightDevice: (stringDeviceIsInArea(DT) = kitchen) ∧
(stringDeviceIsInState(light_switch_1) = light_switch_on) ⇒
stringDeviceIsInState(DT) = light_on.
```

4.2.9 Save

The save button is used to save the current workspace and translate it to FO(·). It takes all the blocks currently in the workspace and sends them to the back-end. After the blocks are parsed into FO(·) code, another process is started where the Interactive Consultant of IDP-Z3 is started with the FO(·) description is already loaded. No error detection or correction is done during the saving, since the IDP-Z3 engine provides a way to explain errors.

4.3 FO(·) parsing and IDP-Z3 integration

For the generating of the FO(·) code we used Python, done locally on the user's machine. When the user saves their constructed KB, the blocks translated each to a JSON-like struc-

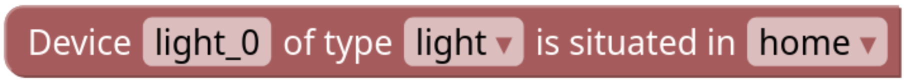


Figure 4.10: An example of a blockly block that creates a new device

ture, when they are all parsed, they are sent to the back-end which translates them into valid FO(·) rules. An example of how the JSON-like structure is parsed is shown below.

```
__NEW_DEVICE__{{
    "deviceName": "light_0",
    "deviceType": "light",
    "deviceArea": "home"
}}
```

Every JSON-like structure has the same format having a function name, followed by a JSON with all the parameters in it.

In this case the block shown in Figure 4.10 would, by the generator in Blockly, be parsed into the above JSON-like structure. It is then translated into the following FO(·) code:

```
vocabulary V {
    type StringDevice := {light_0}
    type lightDevice := {light_0} <: Device
}
structure S:V {
    deviceIsInArea := {light_0 → home}.
}
```

After parsing every block to valid FO(·) code, it is written to a file, after which the Interactive Consultant of IDP-Z3 is started. This is done in a separate process keeping the blockly server running. This allows the user to correct semantic errors in their knowledge base without having to reset everything.

4.4 Interactive Consultant

The Interactive Consultant is used as a validation tool for the created KB. It helps the user understand their mistakes and correct them and gives the user a way to explore their KB and find any semantic errors. This is a crucial part of the application since, according to the study done by Sheik Murad Hassan Anik et. all [3], error detection and correction is one of the hardest, but most important, parts of a home automation system.

Chapter 5

User study

To test whether or not our created application is in fact user-friendly and is able to achieve the flexibility we wanted, we performed a user study. In this chapter we will elaborate on our approach and discuss results of this study.

5.1 Methodology

We searched for participants who had no, or little, prior knowledge of home automation, programming or FO(·), to mitigate bias. Our testing group consisted of 14 participants. Before going to the actual testing, we asked an external tester to already try the application and give us feedback. This was done so we could change things that were intuitive for us as creators, but unintuitive for them. Based on this feedback, we changed the application a bit and adjusted the created manual.

To guide the experiment, we created a manual (Appendix A.2). This explained what home automation is, and what the intend is of automating your home. It mentions concepts like areas, devices, devicetypes and rules who are all nesessary to understand the application. The manual also gives some examples of how to create these concepts using the editor. These examples do not cover every single block, this way the user can still explore the options available in the editor. It also features a simple example of a light that is controlled by a lightswitch. In this way, the user has a rough idea of how a home should be modeled in the editor. We send the testers the manual some time before the tests, so they can read it at their own pace and get familiar with the concepts and the structure of the application.

After they have read the manual, we asked the testers to model some simple homes in the form of cases. These are small examples of homes that are easy to understand and do not require a lot of knowledge about home automation. Each case is further described in Section 5.2. Similar to the test methodology of SmartBlock [18], the cases ramp up in difficulty the further the tester goes. Once the tester has effectively modeled a case, they were asked to check their created KB in the Interactive Consultant of IDP-Z3. This way they could see if their model was correct and if it was not, they could fix it in the blocks-based editor.

Finally, we asked the participants to fill in a questionnaire. It had some questions about the blockly-part of the application, the IC-part, whether or not it was easy to use and whether or not they felt confident in their ability to create a home automation system with this editor and their aquired knowledge. Below are some examples of the type of questions.

- How good is your IT-knowledge?
- Did you work with a blocks-based editor before?
- Were the blocks too complex?

5.2 Cases

In this section we will discuss the cases we used in our study.

5.2.1 Case 1

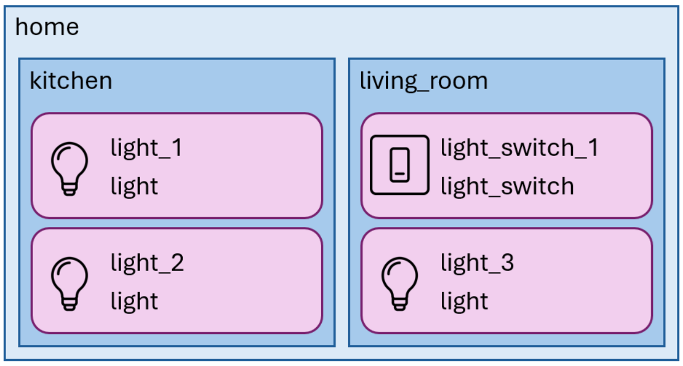
Case 1, shown in Figure 5.1, is the simplest case, with no real extra complexity from the example given in the manual. It is to let the user use the blocks and workflow for the first time. This was also done in the SmartBlock [18] application. The case consists of 3 lights and a lightswitch in a living room and a kitchen, as shown in Figure 5.1a. The rules that go with this case are:

- "light_3" should turn on if "light_switch_1" is on.
- "light_1" and "light_2" should turn off if "light_switch_1" is turned on.
- "light_1" and "light_2" should turn on if "light_switch_1" is turned off.
- "light_3" should turn off if "light_switch_1" is off.

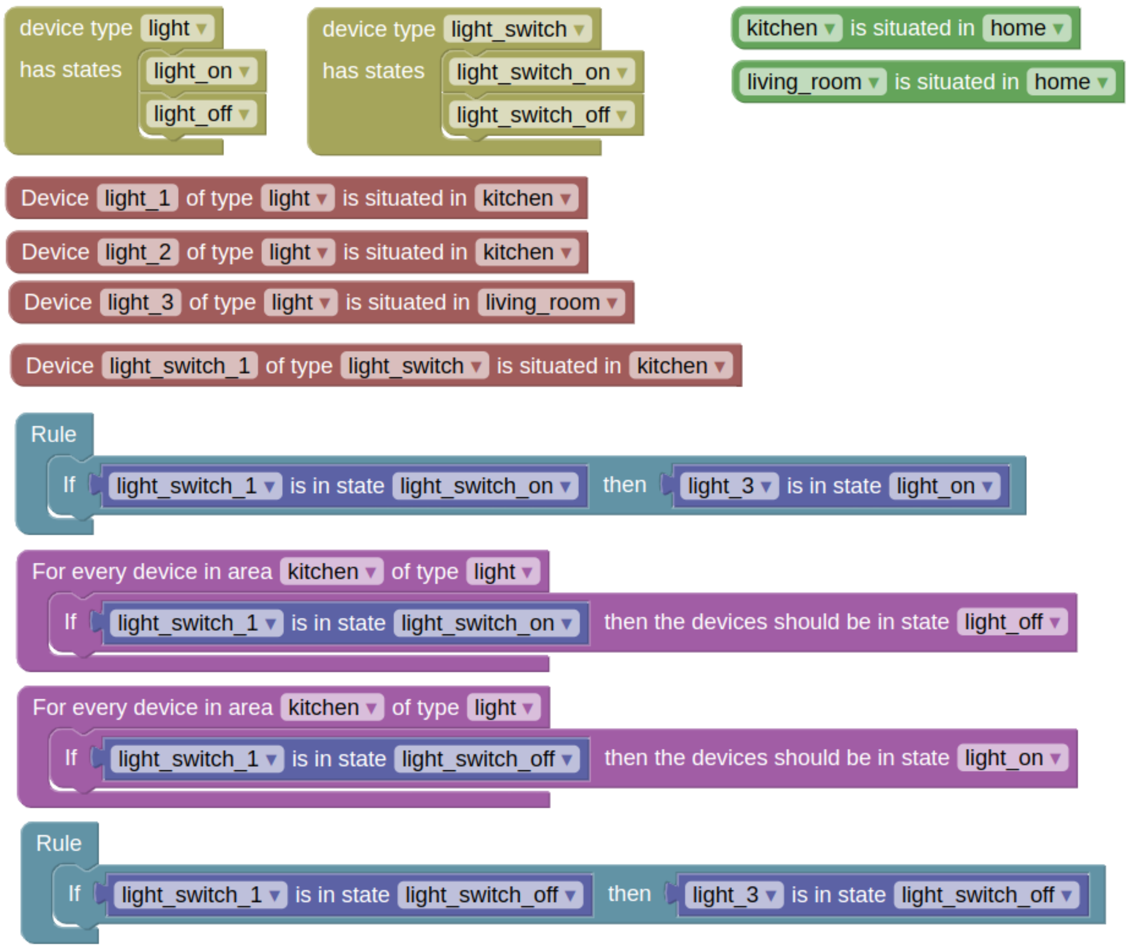
A possible solution for this case in our editor is shown in Figure 5.1b. It is important to point out that this solution is not the only one possible. Most of the users used the "and" block to control the lights in the kitchen.

5.2.2 Case 2

The second case, shown in Figure 5.2, is a bit more complex compared to the first case. It features "stringDevices" as well as "numberDevices". This makes the user experiment with this new type of device and how to model it in out system. The rules are still relatively simple, in the sense that they still only require a simple "if-then". However, they require the



(a) Structure of the home of case 1


device type light
has states light_on light_off
device type light_switch
has states light_switch_on light_switch_off
kitchen is situated in home
living_room is situated in home
Device light_1 of type light is situated in kitchen
Device light_2 of type light is situated in kitchen
Device light_3 of type light is situated in living_room
Device light_switch_1 of type light_switch is situated in kitchen
Rule
If light_switch_1 is in state light_switch_on then light_3 is in state light_on
For every device in area kitchen of type light
If light_switch_1 is in state light_switch_on then the devices should be in state light_off
For every device in area kitchen of type light
If light_switch_1 is in state light_switch_off then the devices should be in state light_on
Rule
If light_switch_1 is in state light_switch_off then light_3 is in state light_switch_off

(b) A possible solution for case 1

Figure 5.1: Case 1

user use rule-blocks that go beyond the blocks used in the guide, the introduction of time is an example of a new block they now need. The case consists of 3 lights, 2 motion sensors, 1 temperature sensor and a fan, these last 2 types are "numberDevices". The case is shown in Figure 5.2a. The rules that go with this case are:

- "light_3" should turn on if "motion_sensor_1" or "motion_sensor_2" is active.
- "fan_1" should turn on to 80 if "temperature_sensor" is 30.
- "light_1" and "light_2" should turn off if the time is 8.

A possible solution for this case in our editor is shown in Figure 5.2b. Just as in case 1, this is not the only solution.

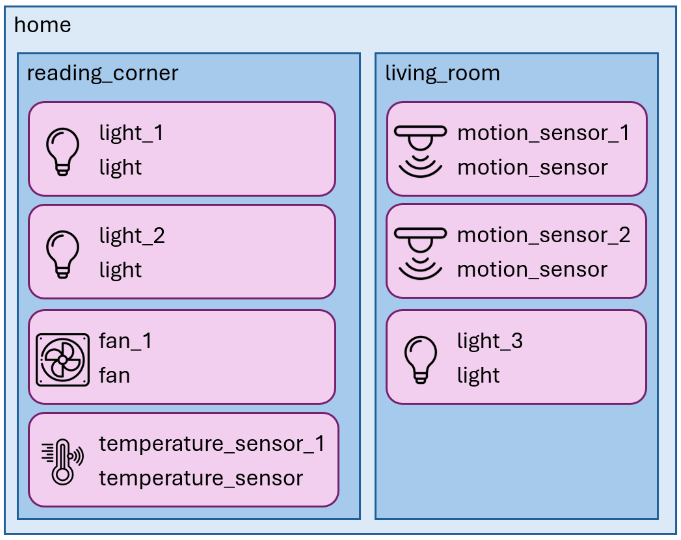
5.2.3 Case 3

In the last case, shown in Figure 5.3, the user was asked to model the most complex home of the three cases. It requires more difficult rules, making use of everything the user has learned so far. The case consists of 3 lights, 2 lightswitches, 1 alarm, 1 blind, 1 motion sensor, 1 variable light and a variable light switch. The case is shown in Figure 5.3a, as is a possible solution in Figure 5.3b. The rules that go with this case are:

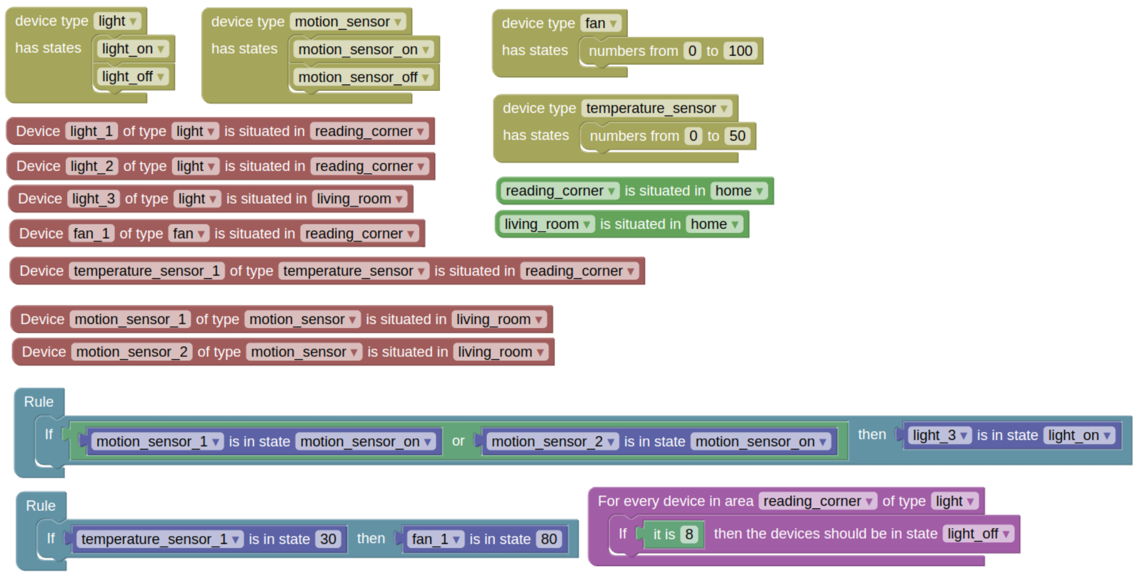
- "light_1" and "light_2" should turn on if either "light_switch_1" or "light_switch_2" is on, but not both. Think of it as lightswitches on different floors that control 1 light.
- all the lights of type "light" should turn on and the alarm should turn on, if the time is 2 and "motion_sensor_1" is in the state "motion_sensor_on".
- "variable_light_switch_1" controls the state of "variable_light_1".
- "variable_light_1" should turn off (value = 0) and "light_3" should turn on if it the time is 4.
- "variable_light_1" should have a value of 40 and "blind_1" should be half_opened, if the time is 6.

5.3 Results

In this section, we will discuss the results of the user study. First, we will present our findings during the modeling of the cases. After that, we will go over the results of the questionnaire and discuss what these results mean for our application.



(a) Structure of the home of case 2



device type light
has states light_on light_off

device type motion_sensor
has states motion_sensor_on motion_sensor_off

device type fan
has states numbers from 0 to 100

device type temperature_sensor
has states numbers from 0 to 50

Device light_1 of type light is situated in reading_corner
Device light_2 of type light is situated in reading_corner
Device light_3 of type light is situated in living_room
Device fan_1 of type fan is situated in reading_corner
Device temperature_sensor_1 of type temperature_sensor is situated in reading_corner

reading_corner is situated in home
living_room is situated in home

Device motion_sensor_1 of type motion_sensor is situated in living_room
Device motion_sensor_2 of type motion_sensor is situated in living_room

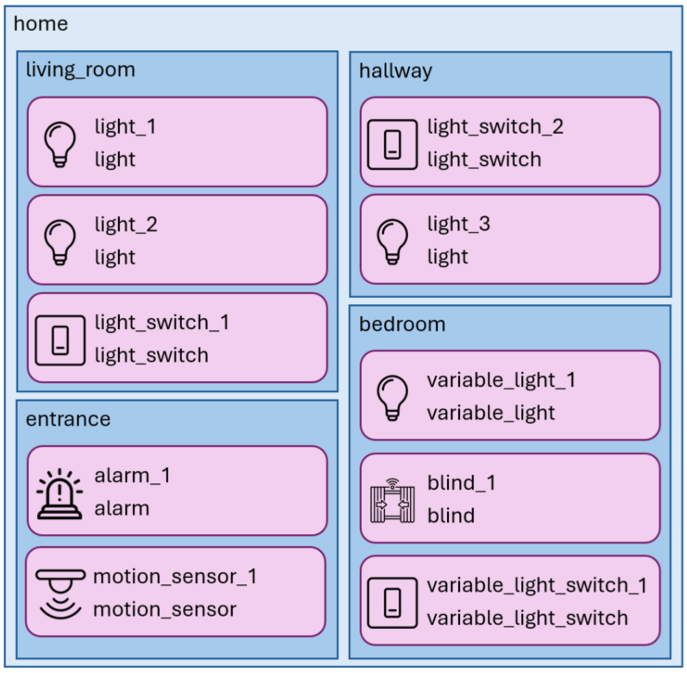
Rule
If motion_sensor_1 is in state motion_sensor_on or motion_sensor_2 is in state motion_sensor_on then light_3 is in state light_on

Rule
If temperature_sensor_1 is in state 30 then fan_1 is in state 80

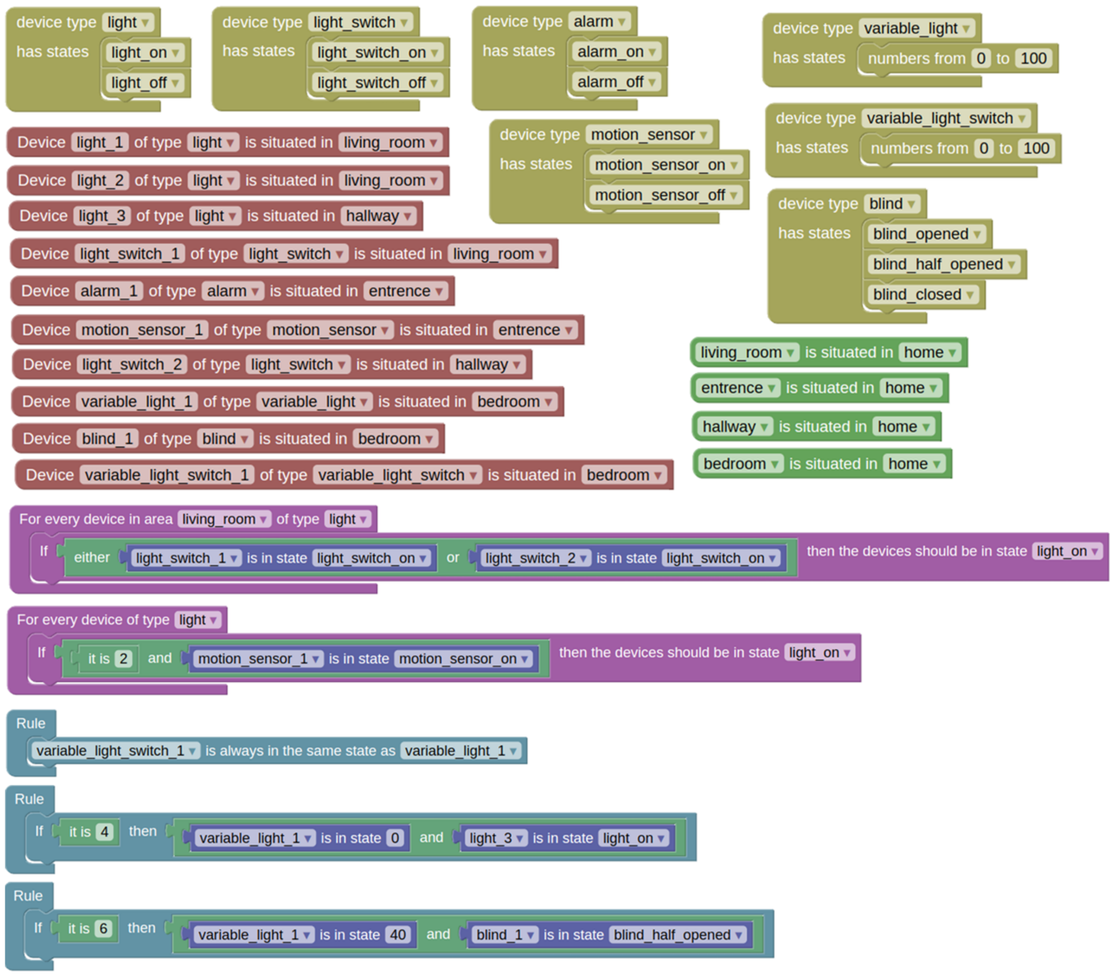
For every device in area reading_corner of type light
If it is 8 then the devices should be in state light_off

(b) A possible solution for case 2

Figure 5.2: Case 2



(a) Structure of the home of case 3



device type light
has states light_on, light_off

device type light_switch
has states light_switch_on, light_switch_off

device type alarm
has states alarm_on, alarm_off

device type variable_light
has states numbers from 0 to 100

device type motion_sensor
has states motion_sensor_on, motion_sensor_off

device type variable_light_switch
has states numbers from 0 to 100

device type blind
has states blind_opened, blind_half_opened, blind_closed

Device light_1 of type light is situated in living_room
Device light_2 of type light is situated in living_room
Device light_3 of type light is situated in hallway
Device light_switch_1 of type light_switch is situated in living_room
Device alarm_1 of type alarm is situated in entrence
Device motion_sensor_1 of type motion_sensor is situated in entrence
Device light_switch_2 of type light_switch is situated in hallway
Device variable_light_1 of type variable_light is situated in bedroom
Device blind_1 of type blind is situated in bedroom
Device variable_light_switch_1 of type variable_light_switch is situated in bedroom

living_room is situated in home
entrence is situated in home
hallway is situated in home
bedroom is situated in home

For every device in area living_room of type light
If either light_switch_1 is in state light_switch_on or light_switch_2 is in state light_switch_on then the devices should be in state light_on

For every device of type light
If it is 2 and motion_sensor_1 is in state motion_sensor_on then the devices should be in state light_on

Rule
variable_light_switch_1 is always in the same state as variable_light_1

Rule
If it is 4 then variable_light_1 is in state 0 and light_3 is in state light_on

Rule
If it is 6 then variable_light_1 is in state 40 and blind_1 is in state blind_half_opened

(b) A possible solution for case 3

Figure 5.3: Case 3

5.3.1 Modeling the cases

All participants completed the cases while we were present with them. We did not assist the participants with the editor or IC in any way but were available to answer questions if there were any about the cases. The presence of someone also provided us with useful information about the difficulties the participants encountered with the application. One such difficulty was the distinction between devicetypes and the devices themselves. This could be observed when users entered device names into the devicetype button. However, this mostly occurred in case 1 and became less of an issue afterwards. Some participants did not struggle with this at all.

Another observation was the speed at which participants modeled the cases. They spent approximately the same amount of time on each case, around 20 minutes. Since the cases increased in complexity, this indicates that participants improved over time. None of the participants seemed to struggle with the Blockly [14] interface. There were no issues with connecting blocks, deleting blocks, duplicating blocks, navigating the menus and workspace, etc.

Most of the participants made some mistakes during the modeling, however not all of them did. The mistakes they made were mostly related to not selecting the correct states when creating the rules and were easily fixed with the help of the IC. Everyone used the IC to check their constructed KBs which either made them sure of the fact that they did not make any mistakes, or made them aware that there was a mistake.

5.3.2 Questionnaire

The questionnaire was sent to the participants after they had modeled all the cases. It consisted of 14 questions about their IT knowledge, their previous experiences with blocks-based editors, the blocks, and the IC. The full results of the questionnaire can be found in Appendix A.3.

We had a fairly wide range of participants regarding their level of IT knowledge. The scores they gave themselves ranged from 3/10 to 9/10, with most of them around a 7 or 8 out of 10. It is worth noting that all the participants have their own idea of an IT-specialist. For some this is being able to work fluently with excel, while others think it is being able to program. Only 21.4% of the participants had no experience with any blocks-based editor. This is probably due to the fact that blocks-based editors are commonly used in schools to introduce young people to programming. The ages of our testers ranged from 14 to 52 years old.

The participants were asked to rate the complexity of the functionality of the blocks on a scale from 1 to 10. This was done to determine whether the testers found the functionality of each block easy to understand and use correctly. The average score was 8.5 out of 10, which shows that the testers found the blocks' functionality easy to understand and had no issues identifying which block should be used for what. The grouping of the blocks in the menus was rated 9.14 out of 10, indicating that the organization of the blocks in the different menus was easy to follow and made sense.

They were also asked about the complexity of the blocks, in other words, whether the blocks contained too much functionality, making them confusing for users. The average score for that question was 8.43 out of 10, suggesting the blocks were not too complex. We also coloured the blocks in a way that made it easier to see which blocks should be used together. The testers were asked about this, and their answers were divided. Some of them did not notice that the colours had meaning, while others thought it was helpful. There were also participants who noticed it but did not find it helpful. The average score—though due to the wide range of opinions, it should be taken with a grain of salt—was 6.36 out of 10.

The participants were also asked about the IC and how useful it is during the modeling of the cases. They gave it a 9.36 out of 10, which further demonstrates the usefulness of the IC. They found it very helpful to see whether their model behaved as expected and to change blocks if there were any mistakes. Not all participants made a mistake in their model (5 out of the 14 did not), but those who did found the IC to be a very helpful verification tool. They gave it a score of 7.67 out of 10 for its usefulness in that regard. The most common mistakes were related to the selection of the correct state while creating the rules. The testers would, for example, model that a light should be in the state "light_on" when a lightswitch is in the state "light_on", which is wrong, since the lightswitch should be in the state "light_switch_on" instead. When the IC did not respond in the way they expected, they were able to identify the rule that caused the issue and fix it.

The testers were asked to rate the overall user-friendliness of the blocks and the IC. The average scores were 8.71 out of 10 for both categories, showing that the testers found the application easy to use overall. As a final question, they were asked whether they felt confident in their ability to create a home automation system using this application after the small introduction through the cases. Their confidence was rated at an average of 8.42 out of 10.

An overview of the results of the questionnaire can be found in Table 5.1.

5.3.3 Discussion

The results of the user study show that our application is quite user-friendly. The participants did not have much trouble using the application and modeling the cases. The colours of the blocks were not as helpful as we had hoped, but they did assist some users. The IC was rated very highly in terms of usefulness, which aligns with the findings on rule validation from the SmartBlock [18] paper. The fact that five users did not make any mistakes during the modeling—despite having no prior knowledge of home automation—is another sign that the application is user-friendly. Those who did make mistakes were able to find them with the help of the IC.

Question	Average score (out of 10) ± standard deviation
IT-knowledge	6.07 ± 1.49
Easy functionality of the blocks	8.5 ± 1.02
Grouping of the blocks in the menus	9.14 ± 1.10
Functionality of the blocks was not too much	8.43 ± 1.28
Usefulness of the colours	6.36 ± 3.00
Usefulness of the IC	9.36 ± 0.84
User-friendliness of the blocks	8.71 ± 1.14
User-friendliness of the IC	8.71 ± 1.20
Confidence in creating a home automation system	8.42 ± 1.28

Table 5.1 Results of the questionnaire

An other interesting finding is that there is not really a difference in the people who already had some experience with a blocks-based editor and those who did not. In fact, 2 of the 3 testers without prior experience with blocks-based editors did not make a single mistake. We also do not see a relation between the IT-knowledge of the testers and their ability to model the cases correctly. The average IT-knowledge score was 6.07 out of 10 overall, while the average IT-knowledge score for the testers who made mistakes was 5.78 out of 10 (Appendix A.3). Considering the standard deviation of 1.49, this indicates that the testers who made mistakes were not significantly less knowledgeable in IT than those who did not.

All these results need to be taken with a grain of salt, as the number of testers was not large enough to draw definitive conclusions. However, it is a good indication of how the application is perceived by users.

Limitations of the application A limitation of our small case size is that the application was not tested on larger homes. Therefore, we cannot make any claims about the scalability of the application. One issue we did encounter is that, since the devicetypes, areas, and states are created using a button, the user cannot access or edit them afterwards. This was not a major issue because the cases were small, but if the application were to be used for full homes, this could become problematic.

At the moment, the application can only be used to create a home automation system. The application is only a modeling tool; it does not offer any means to control devices or areas outside of the IC. If this were to become a real application, IDP-Z3 would need to run regularly to detect changes in the home, so that the structure part of the FO(·) description can be updated to reflect the new home situation.

Currently, it is not possible to create every complex rule. However, this is more a limitation of the available blocks than of the expressiveness of the FO(·) language. For example, blocks like "greater than" are not yet available but are not difficult to add to the application. The definition of time is also not ideal, as it is currently just a number. This could be improved by adding a few blocks for repeated events, such as "every Friday at 8 a.m." This would enable users to create rules that would definitely be useful when modeling an actual home.

There is also no way to create a device that has both numbers and strings as states, due to the vocabulary we chose in Section 3.2. It is possible that other vocabularies would allow for this added expressiveness, but we did not find any that support this while keeping the system simple enough to be modeled in a blocks-based editor.

Chapter 6

Conclusion

In this thesis we created an application for home automation using a first order logic language in combination with a reasoning engine. We identified previous issues with home automation modeling and tried to adress them using the formal language FO(·) and the IDP-Z3 system. Given that this language is not easy to learn and understand, we explored alternative options for user interaction. We landed on a blocks-based editor as the most effective for this goal. Following this, we thought about the way the FO(·) code should be structured and created a blocks-based editor that allows the user to model their own knowledge bases / home automation systems. After the creation of the knowledge base, the user could check whether or not their constructed KB was correct using the Interactive Consultant of IDP-Z3. During the creation of this editor we kept a close eye on the user-friendliness and the different rules a user would want to be able to model. After the creation of the application, we did a user study to test whether or not we succeeded in our goal of creating a user-friendly application while keeping the expressiveness of the FO(·) language and the advantages of the IDP-Z3 system. These tests showed that our application is indeed considered fairly user-friendly. The testers had a great feeling for the functionality and structure of the application while getting better and better throughout the presented cases. The validation tool, aka the Interactive Consultant, helped the users test their KB and correct any mistakes they made. The confidence of our test group in their ability to create a home automation system using our application was high, even though they had no prior knowledge of home automation. While our group was small, it is a good indication of the user-friendliness of our application.

6.1 Future work

This work is certainly not finished, and there is still lots of work to be done. The application can not stand on its own, in the sence that it does not allow to connect with actual physical devices. However, this is crucial if it were to be used in a real home automation setting. The communication with the hardware in a home is however already done well in Home Assistant [1], it would be great if this application could be something that is integrated into the HA system. The problems the testers had with the difference between devicetypes and devices would automaticially be solved by this, since HA can already figure out the states of many physical devices.

The blocks-based editor could also be improved by adding more blocks, such as the "greater than" block, which would allow for more complex rules to be created. The definition of time could also be improved by adding blocks for repeated events, such as "every Friday at 8 a.m." This would enable users to create rules that would now be tedious or even impossible to create.

Regarding the FO(·) backbone, it would be great if more vocabularies would be explored. The current vocabulary works well, but is definitely not the only one or the best one. It would be interesting to see if other vocabularies would allow for more expressiveness, such as the ability to create devices that have both numbers and strings as states, while keeping the system simple enough to be modeled in a blocks-based editor. Even the possibility of other languages could be explored, such as SWRL as done in [16].

The user study also needs to be expanded. While the current study shows that the application is user-friendly, it was done with a small group of testers. A larger group would provide more reliable results and would probably reveal additional insights into the user experience.

Bibliography

[1] *Home assistant*. [Online]. Available: `https://www.home-assistant.io/`.

[2] *Home assistant demo*. [Online]. Available: `https://demo.home-assistant.io/%5C#/lovelace/home`.

[3] S. M. H. Anik, X. Gao, H. Zhong, X. Wang, and N. Meng, *Automation configuration in smart home systems: Challenges and opportunities*, 2024. arXiv: `2408.04755 [cs.SE]`. [Online]. Available: `https://arxiv.org/abs/2408.04755`.

[4] P. Carbonnelle, S. Vandevelde, J. Vennekens, and M. Denecker, *Idp-z3: A reasoning engine for fo(.)* 2022-02-01.

[5] M. Denecker, J. Vennekens, M. Garcia de la Banda, and E. Pontelli, *Building a knowledge base system for an integration of logic programming and classical logic*, eng, 2008-01-01.

[6] *Ku leuven*. [Online]. Available: `https://interactive-consultant.idp-z3.be/`.

[7] M. Deryck, J. Vennekens, J. Devriendt, and S. Marynissen, "Legislation in the knowledge base paradigm: Interactive decision enactment for registration duties", pp. 174–177, 2019. doi: `10.1109/ICOSC.2019.8665543`.

[8] O. M. Group, *Decision model and notation*, 2020.

[9] J. V. Simon Vandevelde Bram Aerts, *Tackling the dm challenges with cdmn: A tight integration of dmn and constraint reasoning*, eng, 2021.

[10] T. Kuhn, "A survey and classification of controlled natural languages", eng, *Computational linguistics - Association for Computational Linguistics*, vol. 40, no. 1, pp. 121–170, 2014, issn: 0891-2017.

[11] N. E. Fuchs, K. Kaljurand, T. Kuhn, M. Marchiori, A. Polleres, J. Maluszynski, P. A. Bonatti, S. Schaffert, C. Baroglio, M. Marchiori, S. Schaffert, A. Polleres, C. Baroglio, J. Maluszynski, and P. A. Bonatti, *Attempto controlled english for knowledge representation*, eng, Germany, 2008.

[12] *Mit*. [Online]. Available: `https://scratch.mit.edu/`.

[13] M. Jonckheere, M. Denecker, G. Janssens, and K. L. F. I. O. M. in de ingenieurswetenschappen. Computerwetenschappen (Leuven) degree granting institution, *A structured, block-based editor for fo(.), with translations to idp-z3 syntax, ast and nl*, eng, Leuven, 2021.

[14] *Google*. [Online]. Available: `https://developers.google.com/blockly`.

[15] S. Kumar and M. A. Qadeer, "Application of ai in home automation", *International Journal of Engineering and Technology*, vol. 4, no. 6, p. 803, 2012.

[16] "Ontology-based expert system for home automation controlling", eng, in *Proceedings of the 23rd international conference on Industrial engineering and other applications of applied intelligent systems - Volume Part I*, Berlin, Heidelberg: Springer-Verlag, 2010, pp. 661–670, isbn: 3642130216.

[17] D. L. McGuinness, F. Van Harmelen, *et al.*, "Owl web ontology language overview", *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.

[18] N. Bak, B.-M. Chang, and K. Choi, *Smart block: A visual block language and its programming environment for iot*, 2020. doi: `https://doi.org/10.1016/j.cola.2020.100999`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S2590118420300599`.

[19] L. Chen, C. D. Nugent, M. Mulvenna, D. Finlay, X. Hong, and M. Poland, "A Logical Framework for Behaviour Reasoning and Assistance in a Smart Home", *International Journal of Assistive Robotics and Mechatronics*, vol. 9, no. 4, pp. 20–34, Dec. 2008.

Appendix A

Appendix

A.1 The application

All the code for the application can be found on our GitHub page: `https://github.com/ThijsAlens/homy`

A.2 The manual

Gebruiksaanwijzing Homy

In deze handleiding wordt beschreven hoe Homy werkt en hoe het best gebruikt wordt.

Introductie

Homy is een systeem dat home automation makkelijk zou moeten maken. Home automation, is zoals het begrip doet vermoeden, een huis automatiseren, bijvoorbeeld automatisch de gordijnen open doen als de zon is opgekomen.

Elk huis bestaat uit “devices”, dit zijn de apparaten van je huis. Bijvoorbeeld een lamp, een schakelaar, een gordijn dat automatisch open en dicht kan gaan… Elk device heeft uiteraard een locatie of “area” in het huis, bijvoorbeeld de keuken, living, slaapkamer… alsook een status of “state” waarin het device zich bevindt. Zo heeft een lamp 2 states: “lamp_aan” en “lamp_uit” en een schakelaar: “schakelaar_aan” en “schakelaar_uit”.

Alle “areas” of locaties in het huis zijn verbonden met elkaar. Zo zijn ze allemaal onderdeel van “home” (het huis zelf). De keuken is bijvoorbeeld onderdeel van de onderste verdieping die op zijn beurt weer onderdeel is van het huis. Dit wordt later duidelijker.

Er moeten uiteraard ook regels zijn die het gedrag van het huis beschrijven. Een voorbeeld van zo’n regel is: “als schakelaar_1 in de state “schakelaar_aan” is, dan moet “licht 1” in de state “licht_aan” zijn”. Verder wordt uitgelegd hoe we deze regels precies moeten modelleren.

Algemene instructies

Homy is een blokjeseditor, dit wil zeggen dat je door blokjes aan elkaar te koppelen een huis kan modelleren. Hierbij zijn een paar dingen belangrijk om te weten vooraleer we in de effectieve blokjes duiken.

De kleuren van de blokjes zijn niet zomaar gekozen voor de esthetiek, er zit een betekenis achter. Blokjes met dezelfde kleur zijn bedoeld om samen gebruikt te worden en kunnen dus een extra indicatie zijn om juist blokjes te koppelen. Dit is een vuistregel die zo goed als altijd geldt.

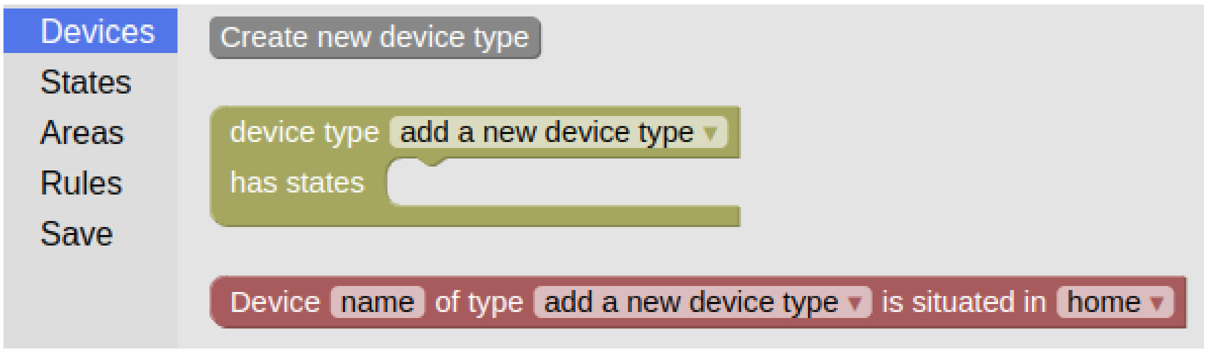
Bij het verbinden van blokjes is het belangrijk goed te kijken of ze wel degelijk geconnecteerd zijn, dit durft al eens mis te lopen. Als een blokje goed geconnecteerd is zul je een geluidje horen. Bij het connecteren van de blokjes, is het belangrijk dat je de connector in de opening plaatst, niet het midden van het blokje.

Wanneer je een nieuw blokje wilt gebruiken of aanpassen moet het eerst uit het menu gesleept worden, aanpassen in het menu is niet mogelijk.

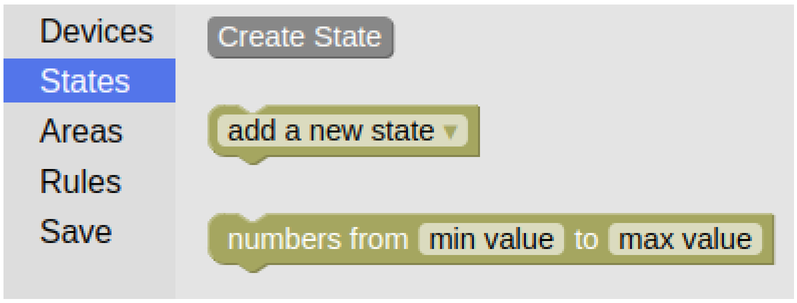
Een blokje met een connectie mag ook nooit alleen staan, er mogen geen open connecties zijn.

Devices

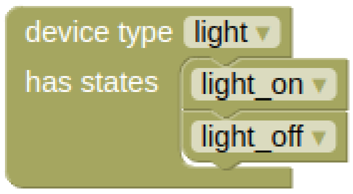
Zoals beschreven in de intro, bestaat een huis uit verschillende devices, maar niet al deze devices zijn verschillend in hun gedrag. Zo gedragen alle lampen in het huis zich op dezelfde manier, namelijk "licht_aan" of "licht_uit". Zo'n groep van devices noemen we een "deviceType". Vooraleer we dus effectieve devices kunnen aanmaken, moeten we deviceTypes aanmaken. Dit gebeurt door de knop bovenaan in het "Devices"-menu in te drukken en het een naam te geven (bijvoorbeeld "light" voor een lamp).


Devices
States
Areas
Rules
Save
Create new device type
device type add a new device type
has states
Device name of type add a new device type is situated in home

Na het creëren van het deviceType moeten we nog beschrijven wat het gedrag is van dat type. Hiervoor maken we gebruik van het geel-achtige blokje onder het Devices-menu. In de dropdown selecteer je het deviceType dat je wenst te beschrijven, in dit geval "light", waarna je states kan toevoegen door het gele blokje onder het States-menu. In dit geval "light_on" en "light_off". Hoe we deze creëren wordt beschreven in het deeltje over States.

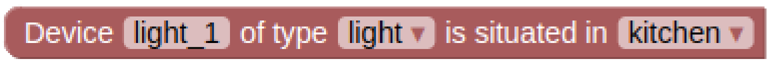


Een compleet deviceType ziet er als volgt uit:



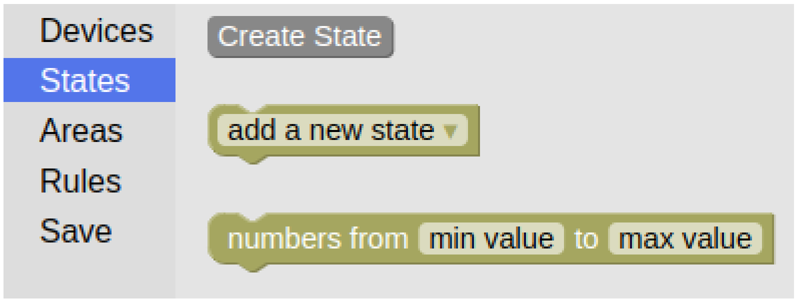
Een deviceType is nog geen echt device, het beschrijft enkel hoe een device van dit type zich moet gedragen.

Om vervolgens de specifieke devices te definiëren, gebruiken we het rode blokje onder het Devices-menu. Hier typ je in het eerste vakje de naam van het device, bijvoorbeeld "light_1". Daarna duid je het correcte deviceType aan, in dit geval "light". Als laatste specifieer je waar het device zich bevindt in het huis, in dit geval "kitchen".



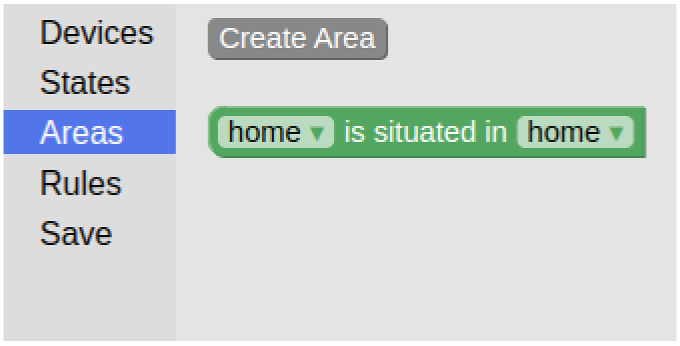
States

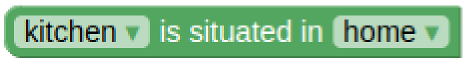
Een state is een mogelijke staat waarin een device kan zijn. Bijvoorbeeld “light_aan” en “licht_uit”. Deze dienen ook aangemaakt te worden, dit kan door, via het menuutje States, op de bovenste knop te drukken en de naam van de state te typen.



Areas

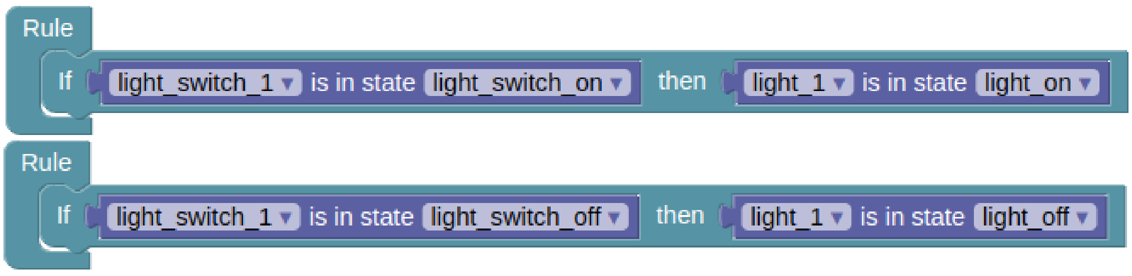
Areas zijn ruimtes / kamers in het huis, bijvoorbeeld de living of slaapkamer. Deze moeten ook aangemaakt worden, dit kan door in het menuutje Areas op de bovenste knop te drukken en de naam van de area te typen. Zoals je kan zien is er nog een groen blokje in het menuutje; dit dient om de hiërarchie van de de areas ten opzichte van elkaar duidelijk te maken. Zo is de keuken deel van het huis.





Rules

De rules of regels zijn een belangrijk onderdeel van het automatiseren van een huis. Alles wat nodig is om dit te doen is te vinden onder het Rules-menu. Een regel gebruikt een als-dan structuur: Als het eerste deel waar is, dan geldt het andere deel. Deze structuur is altijd nodig, dus de eerste 2 blokjes gaan bijna altijd in elkaar staan. In het als-dan-blokje gaat vaak het derde blokje staan dat iets zegt over een device dat in een bepaalde state is, bijvoorbeeld “light_1” is in state “light_on”. Dit is voldoende om basisregels te maken. Het is in sommige situaties echter handiger / sneller / leesbaarder wanneer andere blokjes uit het rules-menu gebruikt worden. Hieronder vind je een voorbeeld van een regel die een lichtschakelaar laat bepalen of het licht al dan niet aan is.


Rule
If light_switch_1 is in state light_switch_on then light_1 is in state light_on
Rule
If light_switch_1 is in state light_switch_off then light_1 is in state light_off

Workflow

Hoe modelleer je nu een huis in de editor van nul te beginnen?

Definieer de devicetypes

Begin met het creëren van de devicetypes en hun gedrag.

- Maak een devicetype aan door de knop.
- Gebruik het gele blokje om het gedrag te specifiëren. Let wel, hiervoor moeten eerst de states gemaakt worden door de knop in het state-menu.

Definieer de areas

Maak vervolgens alle areas aan die nodig zijn en koppel ze logisch aan elkaar.

- Gebruik de knop in het areas menu om nieuwe areas aan te maken.
- Gebruik het groene blokje om de relaties aan te geven.

Definieer de devices

Nu hebben we alles om de devices in het huis te definiëren.

- Gebruik het rode blokje in het devices-menu om nieuwe devices te definiëren.

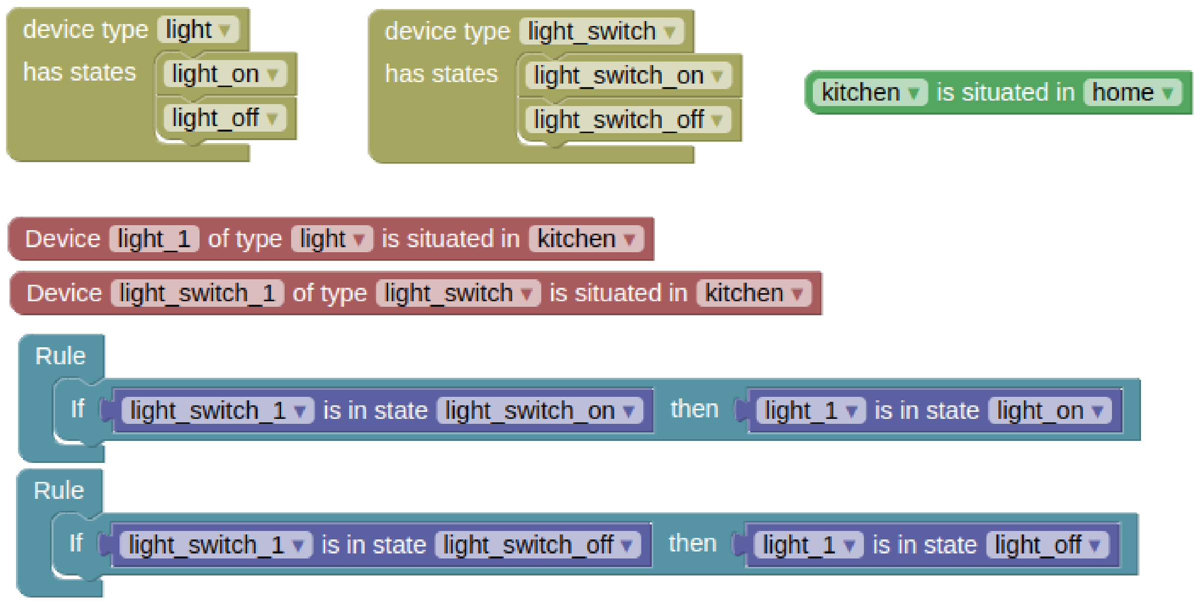
Maak de regels

Als laatste moeten we de regels maken die de devices moeten volgen.

- Gebruik de lichtblauwe (of later rose) blokjes om regels te maken over de devices.
- Wil je complexere regels maken gebruik dan de groen-achtige blokjes.

Uitgewerkt voorbeeld

In volgende screenshot kan je een voorbeeld vinden van een licht dat aangestuurd wordt door een schakelaar in de keuken. Dit is hoe het er ongeveer moet uitzien.



device type light
has states light_on
light_off

device type light_switch
has states light_switch_on
light_switch_off

kitchen is situated in home

Device light_1 of type light is situated in kitchen

Device light_switch_1 of type light_switch is situated in kitchen

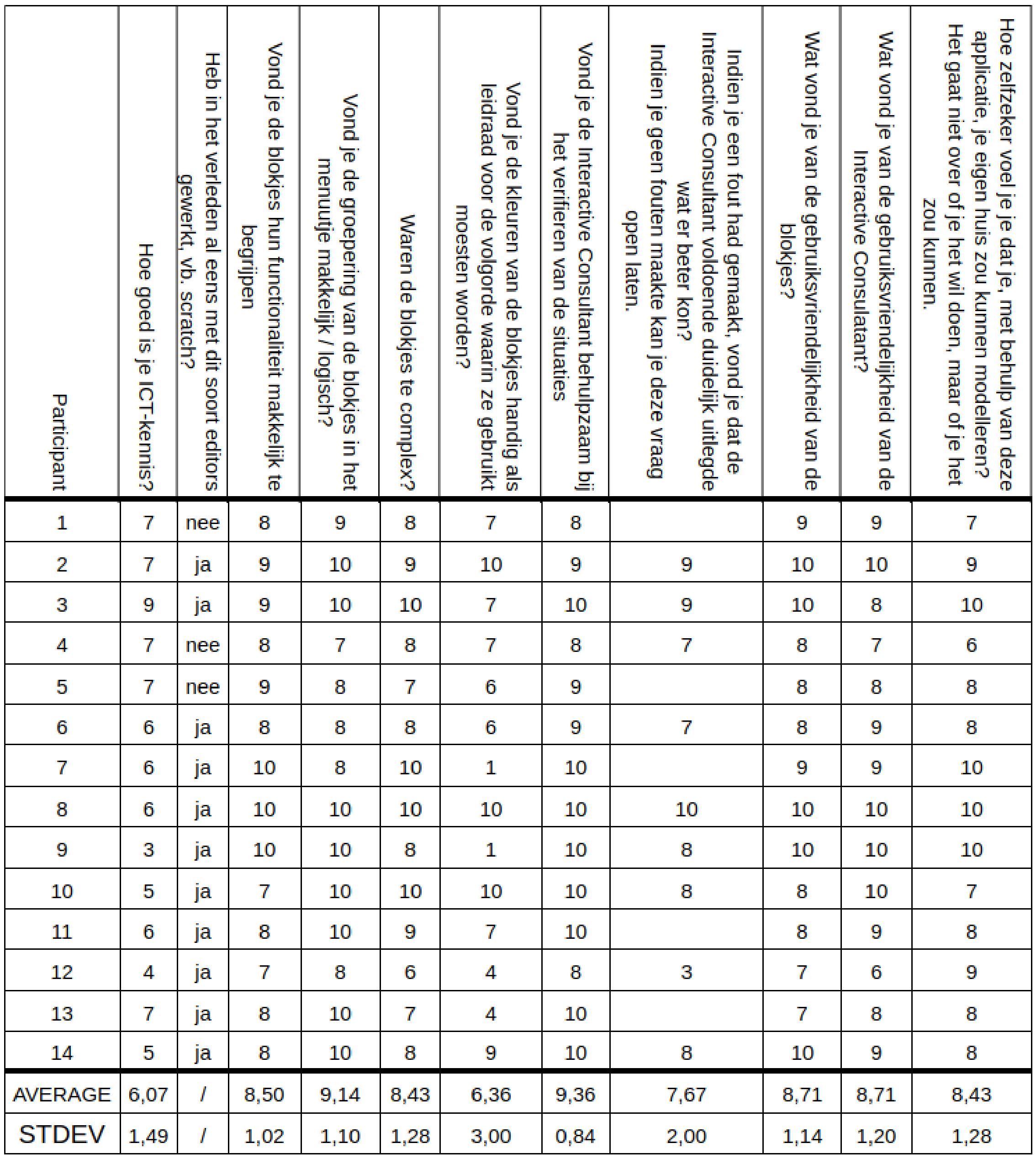
Rule
If light_switch_1 is in state light_switch_on then light_1 is in state light_on

Rule
If light_switch_1 is in state light_switch_off then light_1 is in state light_off

Tot slot

Moesten er nog vragen zijn, vraag het gerust wanneer ik er ben!

A.3 Results of the questionaire



Participant	Hoe goed is je ICT-kennis?	Heb in het verleden al eens met dit soort editors gewerkt, vb. scratch?	Vond je de blokjes hun functionaliteit makkelijk te begrijpen	Vond je de groepering van de blokjes in het menuutje makkelijk / logisch?	Waren de blokjes te complex?	Vond je de kleuren van de blokjes handig als leidraad voor de volgorde waarin ze gebruikt moesten worden?	Vond je de Interactive Consultant behulpzaam bij het verifieren van de situaties	Indien je een fout had gemaakt, vond je dat de Interactive Consultant voldoende duidelijk uitlegde wat er beter kon? Indien je geen fouten maakte kan je deze vraag open laten.	Wat vond je van de gebruiksvriendelijkheid van de blokjes?	Wat vond je van de gebruiksvriendelijkheid van de Interactive Consulatant?	Hoe zelfzeker voel je je dat je, met behulp van deze applicatie, je eigen huis zou kunnen modelleren? Het gaat niet over of je het wil doen, maar of je het zou kunnen.
1	7	nee	8	9	8	7	8		9	9	7
2	7	ja	9	10	9	10	9	9	10	10	9
3	9	ja	9	10	10	7	10	9	10	8	10
4	7	nee	8	7	8	7	8	7	8	7	6
5	7	nee	9	8	7	6	9		8	8	8
6	6	ja	8	8	8	6	9	7	8	9	8
7	6	ja	10	8	10	1	10		9	9	10
8	6	ja	10	10	10	10	10	10	10	10	10
9	3	ja	10	10	8	1	10	8	10	10	10
10	5	ja	7	10	10	10	10	8	8	10	7
11	6	ja	8	10	9	7	10		8	9	8
12	4	ja	7	8	6	4	8	3	7	6	9
13	7	ja	8	10	7	4	10		7	8	8
14	5	ja	8	10	8	9	10	8	10	9	8
AVERAGE	6,07	/	8,50	9,14	8,43	6,36	9,36	7,67	8,71	8,71	8,43
STDEV	1,49	/	1,02	1,10	1,28	3,00	0,84	2,00	1,14	1,20	1,28

Figure A.1: Results of the questionnaire

A.4 Modeling of the rules

The rules described in Section 3.1 actually created in the application.

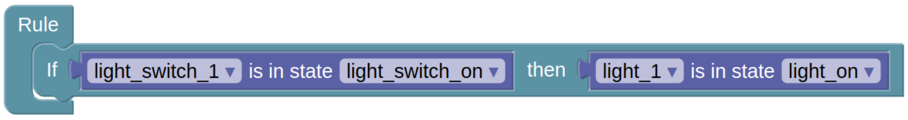

Rule
If light_switch_1 ▾ is in state light_switch_on ▾ then light_1 ▾ is in state light_on ▾

Figure A.2: Rule 1

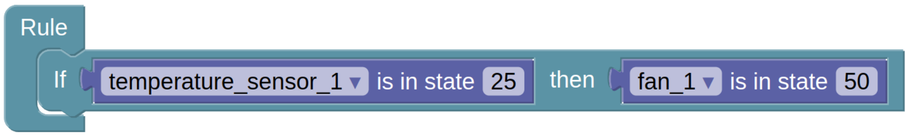

Rule
If temperature_sensor_1 ▾ is in state 25 then fan_1 ▾ is in state 50

Figure A.3: Rule 2

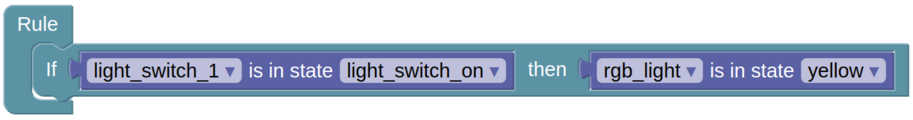

Rule
If light_switch_1 ▾ is in state light_switch_on ▾ then rgb_light ▾ is in state yellow ▾

Figure A.4: Rule 3

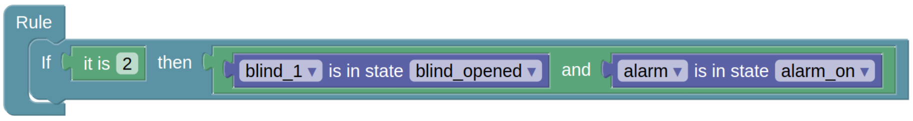

Rule
If it is 2 then blind_1 ▾ is in state blind_opened ▾ and alarm ▾ is in state alarm_on ▾

Figure A.5: Rule 4

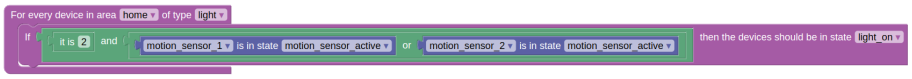
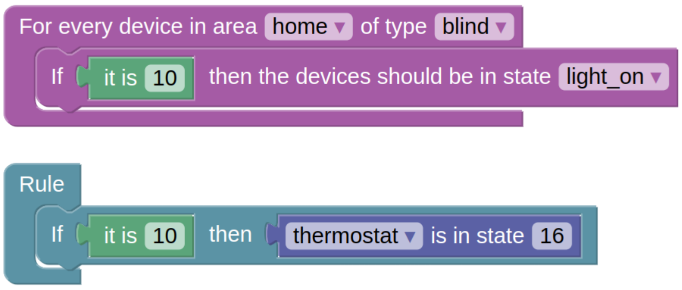
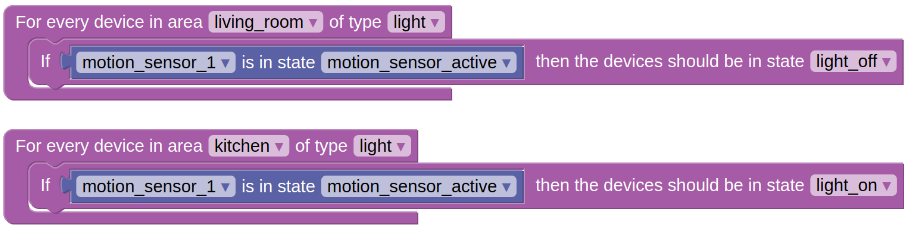

For every device in area home ▾ of type light ▾
If it is 2 and motion_sensor_1 ▾ is in state motion_sensor_active ▾ or motion_sensor_2 ▾ is in state motion_sensor_active ▾ then the devices should be in state light_on ▾

Figure A.6: Rule 5


For every device in area home ▾ of type blind ▾
If it is 10 then the devices should be in state light_on ▾

Rule
If it is 10 then thermostat ▾ is in state 16

Figure A.7: Rule 6


For every device in area living_room ▾ of type light ▾
If motion_sensor_1 ▾ is in state motion_sensor_active ▾ then the devices should be in state light_off ▾

For every device in area kitchen ▾ of type light ▾
If motion_sensor_1 ▾ is in state motion_sensor_active ▾ then the devices should be in state light_on ▾

Figure A.8: Rule 7