KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

DE NAYER (SINT-KATELIJNE-WAVER) CAMPUS

User-friendly Home Automation using IDP

Thijs ALENS

Supervisor(s): S. Vandevelde

Assistant supervisor(s): L. Van Laer

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: Master in de industriële wetenschappen elektronica-ICT

Academic year 2024 - 2025

Contents

Chapter 1

Introduction

1.1 Positioning

We are living in a world where automation is adopted increasing rapidly. 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 define 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 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 (ex. a lightswitch), but it also could have multiple entities (ex. 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

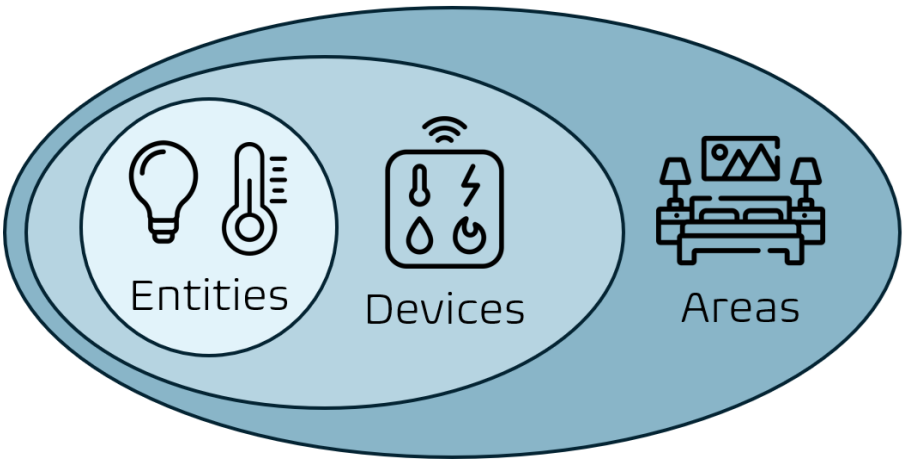

Figure 2.1: The main hierarchy of HA [1]

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, be to 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`

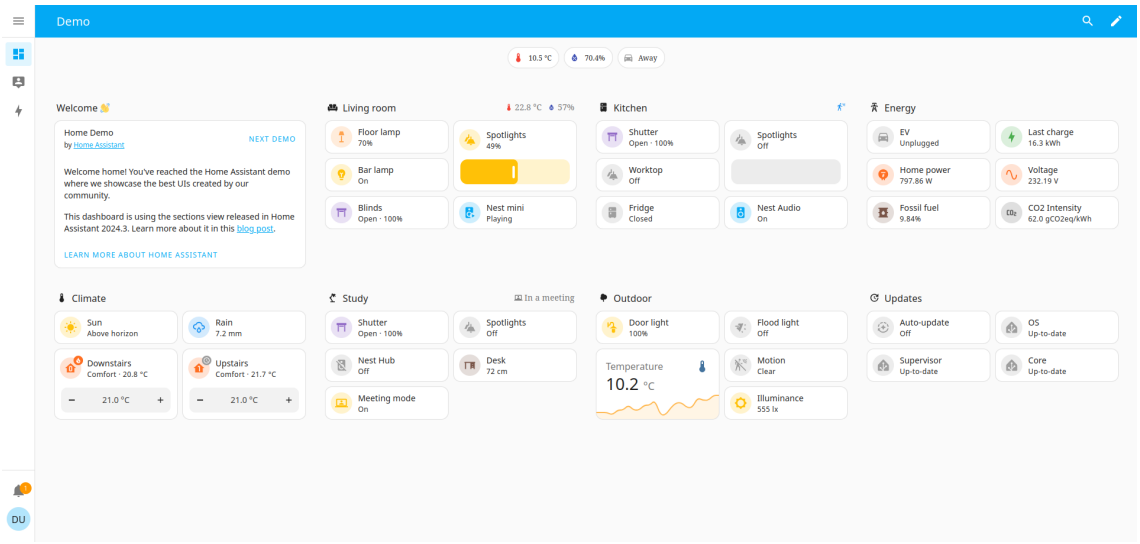

Figure 2.2: An example of a dashboard in HA

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
$\neg$	$\sim$	logical not
$\Rightarrow$	=>	implication
$\Leftrightarrow$	<=>	equivalence
$\forall$	!	for all
$\exists$	?	for at least 1

2.1.3 Issues with the current automation configuration

Home assistant is not perfect. It provides the user with an application that provides a UI for basic home configuration, for more complex ruling, 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 while these tools demonstrate high precision – meaning errors identified are likely genuine (75%-94%) – they suffer from very low recall, identifying only 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 following logic operators $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \forall, \exists$, further described in Table 2.1.

[2] FO(·) is pronounced "Eff-Oh-dot"

The IDP-Z3 language 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}

    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, the value of "State" can only be "On" or "Off". 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 build 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* 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. However, this does not mean that if the left statement is false, the right statement can not be true.

```
∀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.

```
∀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 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 off, then the brightness level is 0 or smaller. A light is 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. When one of the two conditions is true, the other must be false. In other words, the brightness level of a light can either be smaller or equal to 0 or greater then 0.

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. In this case, not all the lights are mapped.

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 build 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 changes, all of these programs would need to be rewritten. This is where 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.

```
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

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 (1) states that a light is of when the brightness level of said light is 0 or smaler. 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

As said before, 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. The user can enter values, after which the IC automatically adjusts the other values, predicates, and functions 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 of the lights from Section 2.2.

As shown in Figure 2.4, the IC explains 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.

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 some values. 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 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, it is useful to explore other options for more accessible notations. In this section, we consider alternative

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

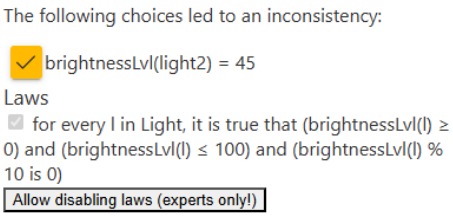

Figure 2.5: An example of how the Interactive Consultant explains why a model can not exist

options in the context of home automation.

2.4.1 DMN

The Decision Model and Notation (DMN) standard [7] 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)

Decision Requirements Diagram ?? laten vallen ?? A DRD is a graph-like structure that represents which decision should be made. It defines how the different decision tables are linked.

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 com-

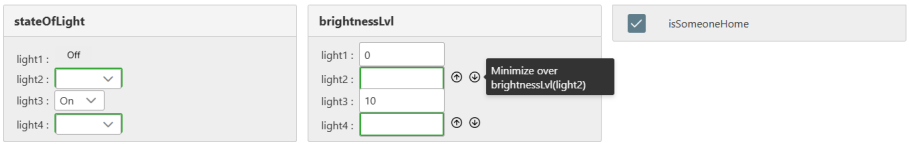

Figure 2.6: An example off how the Interactive Consultant optimizes over something

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

binations 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, its 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" and *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 a FO(·) definition, but not all FO(·) definitions can be represented by a DT. Similarly, implications cannot be represented in a DT.

```
{
   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 [8] is an extension of DMN that addresses the shortcomings of standard DMN by introducing constraint modeling, quantification, types, and functions. Although this makes cDMN more complex, it is far more readable than a complex, standard DMN. 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 [9] (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 [10], 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, written in CNL and shown below, is used as an example. The CNL is likely to make it much 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.

```
theory T:V {
    {
        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 accustomed to, the computer is unable to interpret such input. This challenge, known as the writability problem [9], 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. However, the workspace can become disorganized as projects grow larger.

A well-known example of a blocks-based programming enviroment is Scratch [11], 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(·) [12]. It uses Blockly [13], 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. This is a blocks based editor for the entire FO(·) syntax, this is, however, too complex for what is needed for IoT in home automation as will be further discussed in 3. Still, it provides a good baseline to start from.

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

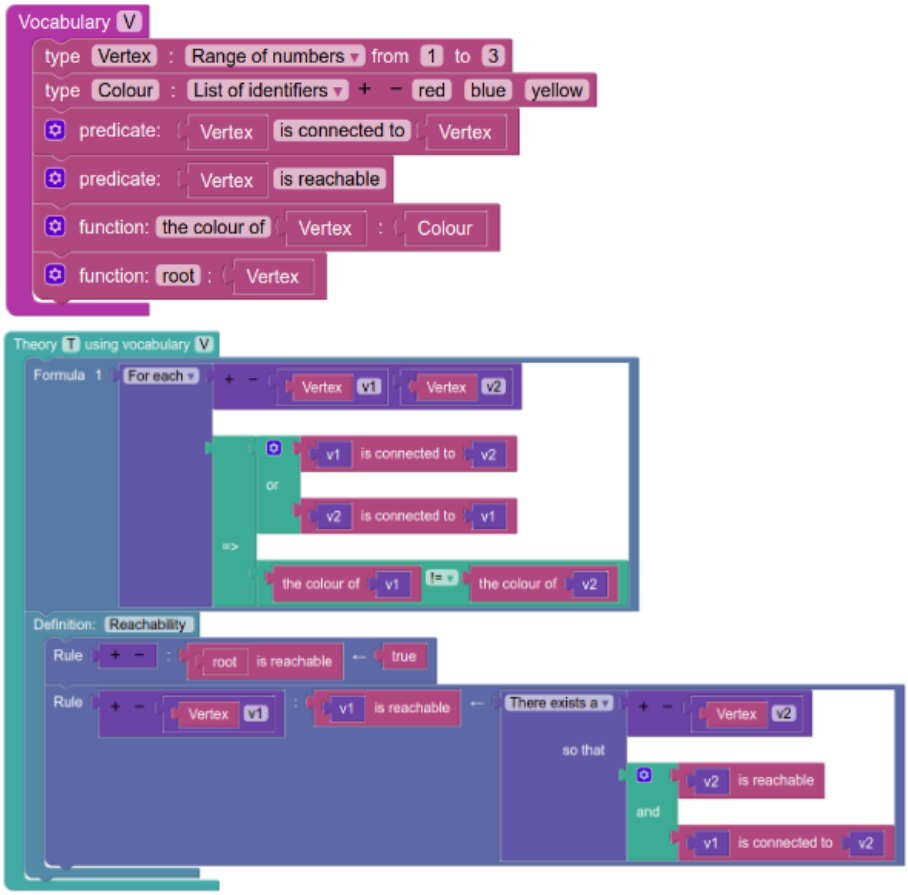

Figure 2.9: An example of a Blockly editor for FO(·)

Table 2.2 Comparison of 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

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 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 pro's and con's 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 [14], 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 sequentially 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. While this can also be automated using AI, that will not be discussed here.

2.5.1 IntelliDomo

IntelliDomo [15] is an ontology-based interface for home automation. It uses an ontology (OntoDomo) to represent the KB, production rules (written in SWRL (Semantic Web Rule Language)) 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. Since SWRL is not the most intuitive language for writing rules for a non-expert, it can be challenging to follow the logic it describes as shown

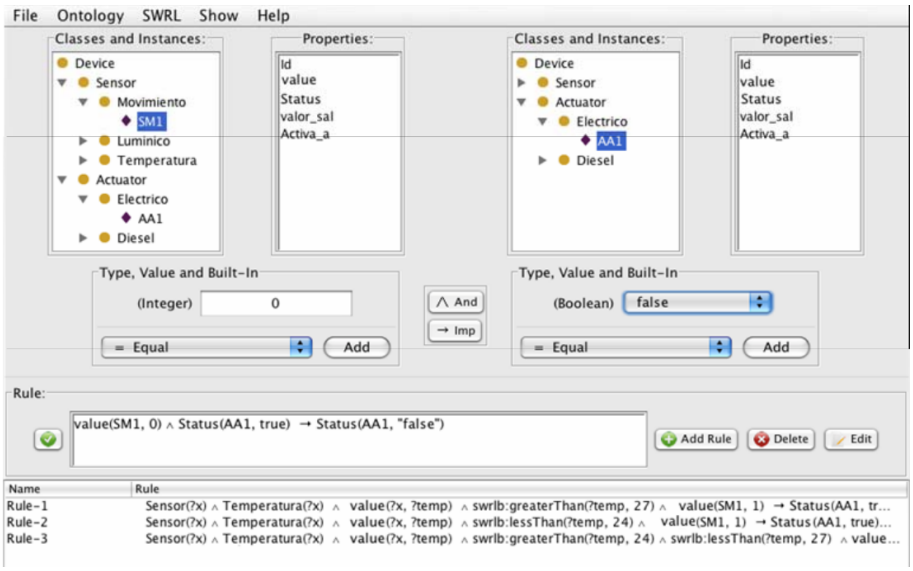

Figure 2.10: The IntelliDomo UI [15]

in 2.10. Once the rules are constructed (the lowest part of the UI), they are difficult to read and understand. 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 2.1.3.

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, 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 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 value 30 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 are not intuitive for the user to fill in.

These papers demonstrate that a knowledge-based approach to home automation works, 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 automating your home.

2.5.3 Event based home automation

Another approach to home automation is an event-based system [16]. 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 system used to achieve this is called Event Calculus. It is a formalism 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

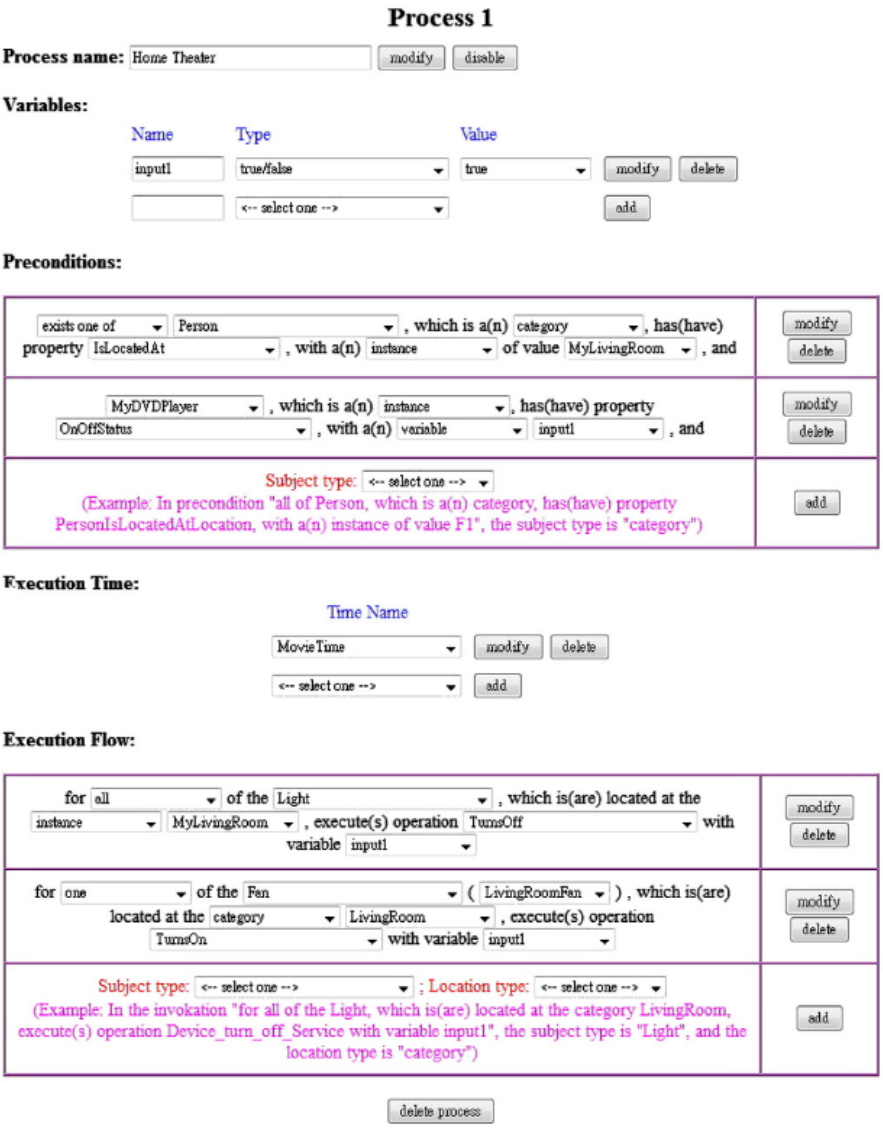

Figure 2.11: The USHAS UI [3]

$$
\begin{aligned}
&Happen(takeTo(kettle, basin), t_0);\\
&Happen(turnOn(tap), t_1);\\
&Happen(add(cold\ water, kettle), t_2, t_3);\\
&Happen(turnOff(tap), t_4);\\
&(Happen(boilWater, t_5, t_6)) \vee (Happen(takeTo(mug, kitchen\ table), t_5));\\
&Happen(takeTo(teabag, kitchentable), t_{51});\\
&Happen(takeTo(milk, kitchen\ table), t_{52});\\
&Happen(takeTo(sugar, kitchen\ table), t_{53});\\
&waterBoiled\ ?\ Happen(add(teabag, mug), t_7);\\
&(Happen(add(boiledWater, mug), t_8));\\
&Happen(add(sugar, mug), t_9);\\
&Happen(add(milk, mug), t_{10});\\
&Happen(stirAround, t_{11}, t_{12});\\
&Happen(takeTo(milk, fridge), t_{13});\\
&Happen(takeTo(sugar, cupboard), t_{14})\\
&Here\ t_0 < t_1 < t_2 < t_3 < t_4 < t_5 < t_6, t_5 < t_{51} < t_{52} < t_{53} < t_6, \wedge t_7 < t_8 < t_9 < t_{10} < t_{11} < t_{12} < t_{13} < t_{14}.
\end{aligned}
$$

Figure 2.12: An example of the Event Calculus code for making a cup of tea [16]

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 2.12. 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.

2.5.4 Smart Block

SmartBlock [17] is a user-friendly blocks-based interface for the SmartThings IoT platform. They built their editor using Blockly [13], a JavaScript framework for creating blocks-based editors. 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.

The user study done by SmartBlock is something we also want to do with our blocks-based editor, it helps to give us an understanding of how our editor did. We probiably would like to change some specifics about the rules and situations, but the diversity of the group and the main objectives they had to do, can stay the same.

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 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.

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 simple enough to be recreated using "simple" blocks. We will discuss the advantages and disadvantages of some 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. We will discuss some example situations that people would want to model. By doing this, the system will be built for the user's convenience, and not necessarily for ease of implementation. By creating situations, we also have a good idea of what the limitations of our system will be.

Devices Device are the cornerstone of any 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 possible states it can be in, for example, a lightswitch can be "On" or "Off". The user could configure these states themselves, but most devices already have a standard set of states. Home Assistant also uses the standard states of the devices to configure them in their UI. This is great, since the user does not need to figure out the possible states of a device themselves. However, to implement this in an editor, we would need to read out the states from Home Assistant, which means we would need to integrate our system into Home Assistant's. This will not be done in this editor.

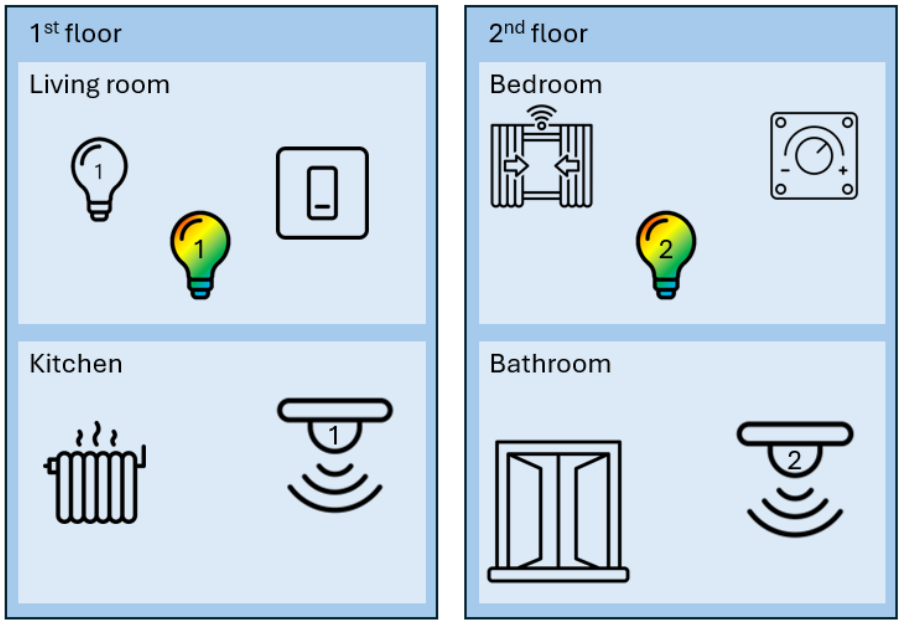

Figure 3.1: An example of a home automation system used to explain the different FO(·) implementations

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 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 the way USHAS does it, where all the areas are semantically connected, as explained in 2.5.2. This fact also enables the user to create rules for super-areas, which also hold for all the areas that fall under it. Since 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.

Rules 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 rules they might want to create, increasing in complexity further down the list.

- Turn on a light when a lightswitch is turned on.
- Set the brightness of a light to 50 when the lightswitch is turned on.
- Set an RGB light to the colour yellow when the lightswitch is turned on.
- Turn on light0 and light1 when the lightswitch is turned on.
- Turn on all the lights in the living room when the lightswitch is turned on.

- Turn off all the lights in the living room and turn on all the lights in the kitchen when the motion sensor in the kitchen is triggered.
- Turn off all the lights on the first floor when a motion sensor on the second floor is triggered.

3.2 Vocabulary and theory

The vocabulary is an important part of the FO(·) specification as it defines the possibilities and limits for the rest of the specification. There are lots of different aproaches possible, where each needs to be effective with the idea of ease of use and effectiveness in a real-world application. The theory, and therefore the possible rules, is directly affected by the choice of vocabulary, so this also needs to be considered. It is also important that the FO(·) specification does not lie to far from the blocks, sinse we want to make use of the IC as a way to verify the constructed knowledge base. If the specification is too different from the blocks, the user will not recognize what blocks are responsible for the errors in the IC.

```
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.
}
```

As a first approach, consider a vocabulary where all the devices are defined as constants. Every device is linked to an area. So when creating a possible model, a device could be "light1 → livingRoom". The problem with this way of representing the devices is that there is no way to group the devices. A rule that states that all the devices in the living room should be turned off is not possible, but is something the user would want. 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. The user also cannot implement any form of grouping in the devices; for example, a rule stating that all the lights should turn off when the lightswitch is off, is not possible with this vocabulary.

An other shortcoming of this vocabulary is the inability to represent relations between areas.

This is because the vocabulary does not allow to group devices, as stated earlier, as well as the inability to group areas. This is also something the user would want to have since it could be a way to structure their home in the application which makes it feel familiar.

```
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 := {ringing, off}
    bellState : Device → BellState
}
```

As a second aproach, 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. A slight drawback of this vocabulary is that IDP-Z3 does not provide much assistance if anything is wrongly defined. It would be preferable if IDP-Z3 could offer more support in preventing user mistakes. It also does not look that pretty in FO(·), since every new device, even if it is of the same type (light1, light2), it needs te be defined with its own type. This means that the user needs to specify, for each device, what states it has. This is not ideal, since the devices like lights, lightswitches, curtains... are pretty frequently used in houses.

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

    deviceIsInArea: Device → Area
    // subArea * superArea -> Bool
    areaIsSubAreaOf: Area × Area →  𝔹
    deviceIsInState: Device → State

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

This is where the third and final vocabulary comes in. 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 does not differ much

from 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.

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 seperation in the devices based on how their state requires them to be defined. This makes it harder to implement behind the scenes, but does not change all that much for 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
        // subArea * superArea -> Bool
        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 tree 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, it is however very easy to create blocks for it. The secod 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, it is still easy to understand while making the device-creation easier.

Chapter 4

Implementation

In this chapter we will discuss what design-dissicions we have made while creating the blocks-based editor. 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 [13], as stated before, this is a JavaScript-based framework for blocks-based editors. It is, at its core, a very flexible framework that allows you to create custom blocks who translate to a code-snippet. 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 aren't 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 who are related to each other, for example the blocks who define an area and the block 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 put 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 menu's inside the blocks so the user can, for example, not select a devicetype when a state is needed.

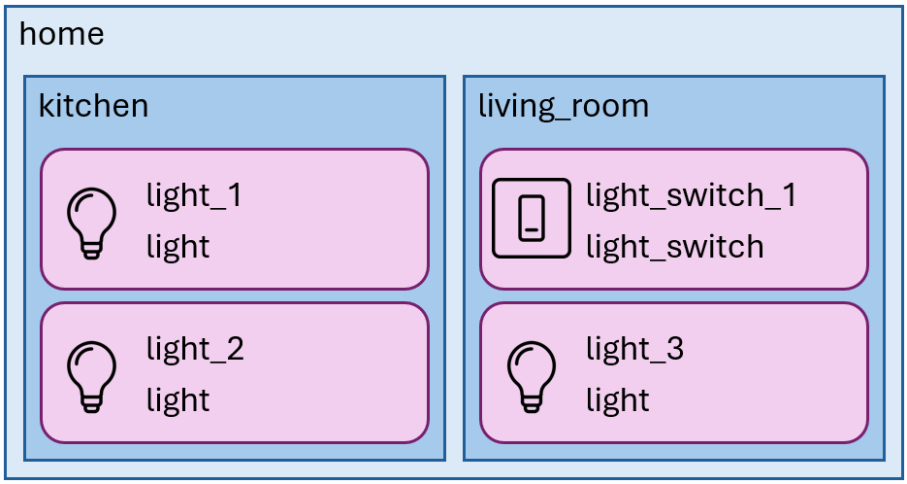

Figure 4.1: An example of a simple home

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. Not all the functionality is done in blocks, there are, for example, buttons to create a new devicetype or a new area. These are different, but will also be discussed here. The blocks will not be discussed in the order they are placed in the editor, to make it easier to understand and follow. The FO(·) descriptions are not complete, in the sense that they will not function without extra vocabulary and/or theory. The focus is 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 4.1.

4.2.1 States

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 name, the state is added to the globaly 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. To define the states of a "numberDevice", the block in Figure 4.2, which speaks for itself, is used. The states are used to define what a device can do, for example a light (in the example light_1, light_2 and light_3) can be in the states "light_on" or "light_off". There is no imediate 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 a state-block should not be used on its own.

Figure 4.2: An example of the blockly block that defines the states of a number device

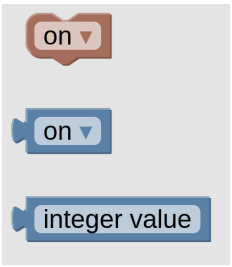

Figure 4.3: The blocks related to the states

4.2.2 Areas

Areas are created in the same way is states, they use a button who 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, both in the vocabulary and structure as shown below.

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

4.2.3 Devicetypes creation

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.

4.2.4 Devicetypes definition

The block shown in Figure 4.5 is used to specify the states of a given devicetype. There are two different types of devicetypes, sinse there are two types of devices, however, this does not reflect in the deviceType block sinse this is already done in the states. Behind

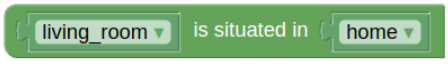

Figure 4.4: An example of the blockly block that defines the relation between two areas

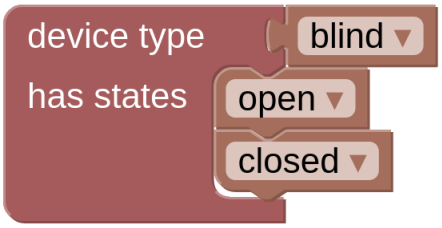

Figure 4.5: An example of the blockly block that defines the states of a device

the scenes, the program looks wether 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 structure as shown below.

```
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.
}
```

The theory part of the FO(·) description is there because light_switchDeviceStates is a subtype of StringState. Sinse no rules restrict the light_switchDevice to only have the states of light_switchDeviceStates, we need to add a rule that states that the device is in one of the states of the devicetype. This is not ideal, but it is a way to make sure that the user can not select a state that does not belong to the devicetype while using the Interactive Consultant.

4.2.5 Devices

A device is created using the 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$''_1 is of the type "light_switchDevice'' and is located in the living_room. The block has a direct$ translation, shown below. However, this again depends on wether it is a number or string device. The line created by the devicetype-block is changed to include the newly defined device.

```
vocabulary V {
    type stringDevice := {light_switch_1}

    type light_switchDevice := {light_switch_1} <: stringDevice
```

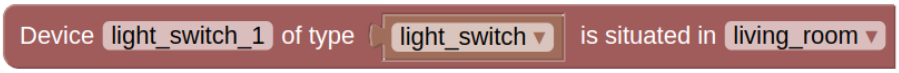

Figure 4.6: An example of the blockly block that creates a new device

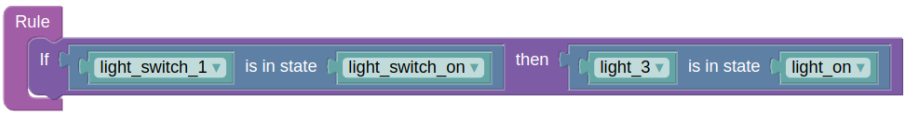

Figure 4.7: An example of the blockly block that creates a single rule

```
}

Structure S:V {
    stringDeviceIsInArea := {light_switch_1→living_room}.
}
```

4.2.6 Single rules

The blocks shown in Figure 4.7 are used to create a standalone rule. The user can create a precondition 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", the "light_3" should be in the state "light_on". The block has a direct FO(·) translation, shown below.

```
theory T:V {
    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. The user selects a device which state dictates wether 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.

```
theory T:V {
    ∀DT in LightDevice: stringDeviceIsInState(light_switch_1) =
   light_switch_off ⇒ stringDeviceIsInState(DT) = light_off.
}
```

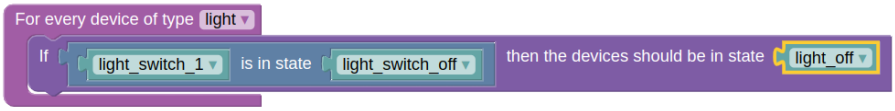

Figure 4.8: An example of the blockly block that creates an enumerating rule

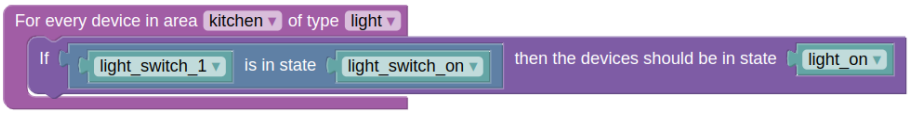

Figure 4.9: An example of the blockly block that creates an enumerating rule over areas

4.2.8 Enumerating 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. After which the user can input one of the previous rules to create a rule that applies only in a sertain 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.

```
theory T:V {
    ∀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 the JSON-like structure and sends a post-request to the flask-server started at launch of the application. After the JSON-like structure is parsed into FO(·) code, another server is started with the Interactive Consultant of IDP-Z3 where the FO(·) description is already loaded. It is important to note that the user does not get corrected on their mistakes in the blocks-based editor, since the IDP-Z3 engine provides a way to explain errors.

4.3 FO(·) parsing and IDP-Z3 integration

For the parsing of the FO(·) code we used python, this is done locally on the user's machine. When the application is started a flask-servver is started which is used to get the generated JSON-like structures from the blocks and parse them into valid FO(·) rules. An example of how the JSON-like structure is parsed is shown below.

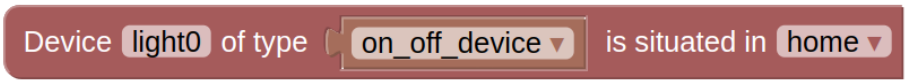

Figure 4.10: An example of a blockly block that creates a new device

```
__NEW_DEVICE__{{
    "deviceName": "light0",
    "deviceType": "  on_off_device",
    "deviceArea": "home"
}}
```

Every JSON-like structure has the same format:

- a function-name (always surrounded with double underscores), in this case NEW_DEVICE
- a JSON structure with the necessary parameters for that function (always encased in double curly brackets)
- a "\n" at the end of the line

In this case the block shown in Figure 4.10 would, by blockly, be parsed into the above JSON-like structure. After which it is translated into the following FO(·) code:

```
vocabulary V {
    type Device := {light0}
    type on_off_device := {light0} <: Device
}
structure S:V {
    deviceIsInArea := {light0 → floor1}.
}
```

After parsing every block to valid FO(·) code, the code is written to a file, after which a server is started that runs the Interactive Consultant of IDP-Z3. By creating a new server, the blockly server keeps running, so if the created KB has some semantic errors, the user can update the blocks accordingly without needing to reset everything.

Chapter 5

User study

To test wether or not our created application is in fact user-friendly and is able to achieve the flexibility we wanted, a user study was done. In this chapter we will walk you through our approach and results of this study.

5.1 Cases

We have constructed 3 cases for the participants to model, increasing in difficulty the further they go. The first case showed in Figure **??** is a very basic case where a light switch controls 3 lights

5.2 Results

Chapter 6

Conclusion

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] "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.

[16] 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.

[17] 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.