KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

DE NAYER (SINT-KATELIJNE-WAVER) CAMPUS

User-friendly House Automation using IDP

Thijs ALENS

Supervisor(s): S. Vandevelde

Assistant supervisor(s): L. Van Laer

Master's Thesis submitted to obtain the degree of Master of Science in: Electronica-ICT programme Software Engineering

Academic year 2024 - 2025

Contents

Chapter 1

Introduction

1.1 Positioning

We are living in a world where automation is 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's 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, specifically 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 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

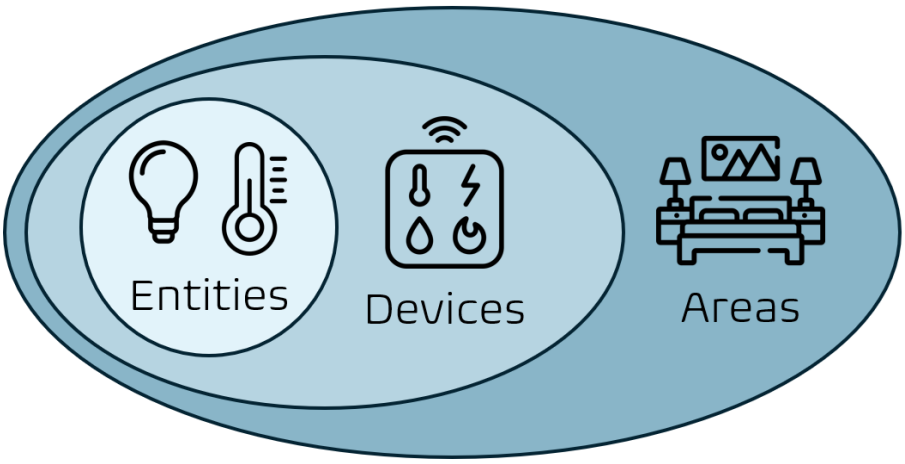

Figure 2.1: The main hierarchy of HA [1]

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 room 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`

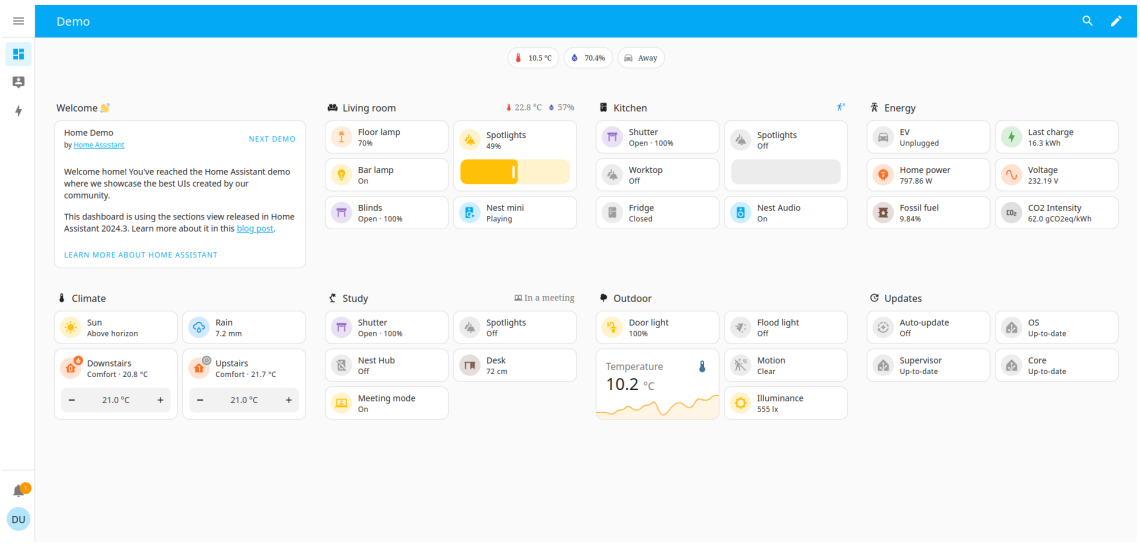

Figure 2.2: An example of a dashboard in HA

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
$\forall$	!	for all
$\exists$	?	for at least 1

2.2 FO(·)

FO(·)[2] (aka FO-dot) is the Knowledge Representation Language used by the IDP-Z3 reasoning engine [3]. It is an extension of first-order logic (FOL), which makes use of the following logic operators $\wedge, \vee, \neg, \Rightarrow, \forall, \exists$, further described in Table 2.1.

The IDP-Z3 language consists of, at a minimum, a vocabulary and a theory. The vocabulary is used to describe which symbols 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(·):

```
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.
```

[2] FO(·) is pronounced "Eff-Oh-dot"

```
    // 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 lights are defined by creating a type called "Light" in the vocabulary. Additionally, a type called "State" is defined, which has two possible values: "On" and "Off". A type, in essence, represents a domain of values, in this example, for instance, a "State" that has 2 values: "On" and "Off". Alongside types, several functions are also defined in the vocabulary. These map one or more inputs to an output.

- "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 formula's in the theory.

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

The first rule ensures that if nobody is home, the lights should be off.
In most cases, this can be interpreted as an "if then" structure. 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. It is used to eliminate any room for misunderstanding in an implication. 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 smaler. A light is on when the brightness level is greater then 0.
This last rule is a definition. It seems the same as an equivalence, but it is not. This is because using a definition explicitly indicates the intention to define the formula on the left-hand side based on the formula on the right-hand side. This sense of direction is less apparent when using an equivalence.

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 [3] 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.

```
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

2.3.1 The Knowledge-Base Paradigm

The IDP-Z3 engine implements the Knowledge Base paradigm [4], 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. Figure 2.3 shows 3 possible models of the lights example.

Propagation The reasoning engine can compute all the logical consequences of a certain rule. 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, this rule is irrelevant:

- Rule 2 states that when nobody is home, the light should be off, making the first part of the right-hand side of the equivalence irrelevant.
- The definition (rule 3), states that a light is "Off" when the brighness-level of the light is 0, making the second part irrelevant.

```
(1) ∀l in Light: stateOfLight(l) = Off  ⇔  (¬isSomeoneHome() ∨ (
   brightnessLvl(l) = 0)).
(2) ∀l in Light: ¬isSomeoneHome() ⇒ stateOfLight(l) = Off.
(3) {
       ∀l in Light: stateOfLight(l) = Off ← brightnessLvl(l) ≤ 0.
       ∀l in Light: stateOfLight(l) = On ← brightnessLvl(l) > 0.
    }
```

2.3.2 Interactive consultant

As said before, the Interactive Consultant [5] (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.

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

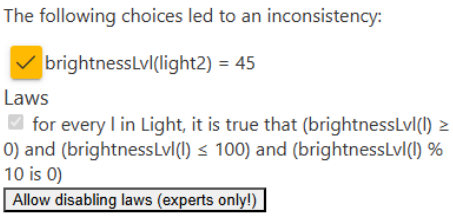

Figure 2.5: An example of how the Interactive Consultant explains why a model can not exist

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 interfaces. In this section, we consider alternative options in the context of home automation.

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

2.4.1 DMN

The Decision Model and Notation (DMN) standard [6] 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 A DRD is a graph-like structure that represents which decision should be made. It includes input fields (rounded boxes) that provide the system with data needed for making decisions. The squared boxes denote decisions.

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, its FO($\cdot$) representation can be found below. It states that a light is "On" if the brightness level is greater then 0 and that it is "Off" if the brightness level is 0. Important to note is that all DTs can be represented by a FO($\cdot$) definition, but not all FO($\cdot$) definitions can be represented by a DT. Similarly, implications cannot be represented in a DT.

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

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

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 [7] 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. If no one is home, the light should be off. The FO(·) code that represents this table is shown below.

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

2.4.2 CNL

A Controlled Natural Language [8] (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 [9], 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 is used in its Interactive Consultant. This thesis will use this 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.

```
vocabulary V {
    type Light
    type State is {On, Off}
    stateOfLight: Light → State
    brightnessLvl: Light → Int
    isSomeoneHome: () →  𝔹
}

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 .
    }
    for all l in Light : not isSomeoneHome() are sufficient
conditions for stateOfLight(l) is Off .
    for all l in Light : (brightnessLvl(l) is greater than 0) ∧
 (brightnessLvl(l) is less than 100) ∧ (brightnessLvl(l) % 10 is
 0) .
    for all l in Light : (stateOfLight(l) is Off) is the same
as (not isSomeoneHome() or (brightnessLvl(l) is 0)) .
}

structure S:V {
    Light := {light1, light2, light3, light4} .
}
```

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 [8], 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 end up learning not only the syntax but also the boundaries of what is and is not possible within the CNL, which can be

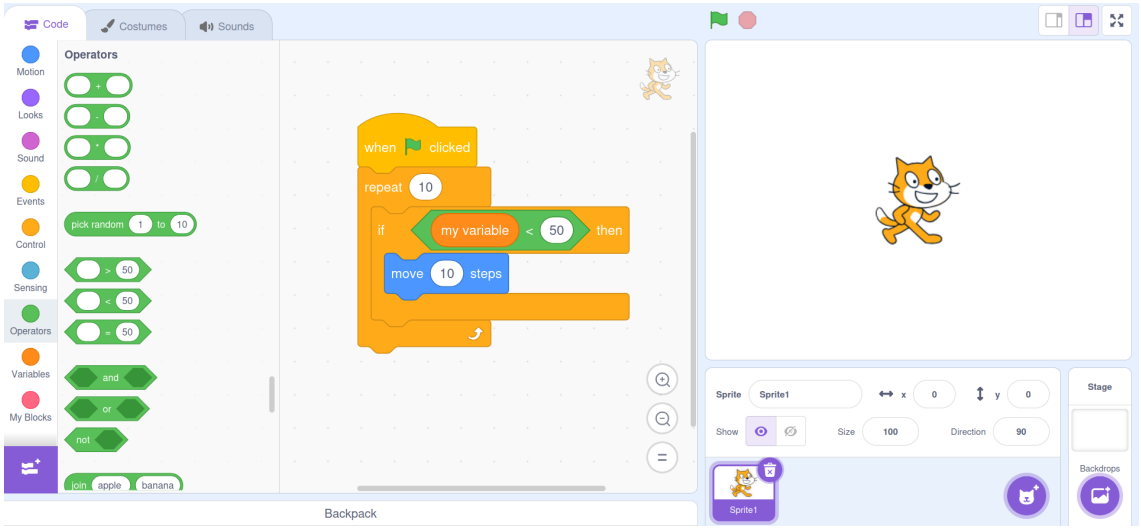

Figure 2.9: An example of the scratch UI

frustrating and counterproductive.

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 [10], 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. In Figure 2.9 an example of a workspace is shown.

FO(·) blocks-based editor There already exists a blocks-based editor for FO(·) [11]. It uses Blockly [12], 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 2.10. This is, however, too complex for what is needed for IoT in home automation. Still, it provides a good baseline to start from.

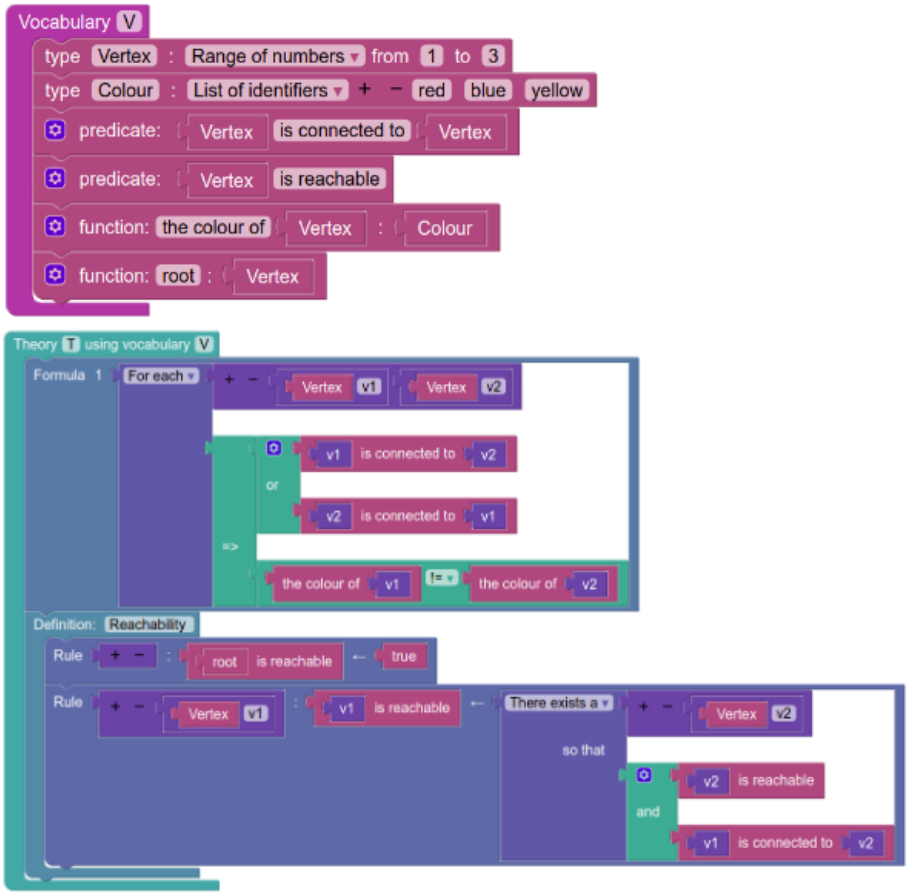

Figure 2.10: An example of a Blockly editor for FO(·)

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

In this chapter we will describe previous works on logical reasoning and home automation. More specificially an example of how knowledge representation is done alternativly. Why the use of YAML files for home automation configuration is not ideal.

2.5.1 IntelliDomo

IntelliDomo [13] is an ontology-based interface for HA. 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, SWRL is not the most intuitive language for writing rules for a non-expert, even when the syntax is pre-constructed. Without familiarity with SWRL, it can be challenging to follow the logic it describes.

This paper demonstrates that a knowledge-based approach to home automation works, but it is important to note that their approach is not the most user-friendly one.

2.5.2 Issues with the current automation configuration

Home automation configuration often relies on YAML files for defining a home's setup, which is how HA manages configurations behind the scenes. If certain automations cannot be configured through the HA UI, users must modify these YAML files themselves. However, YAML is not user-friendly, making it challenging for users unfamiliar with it. 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 [14], 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.

2.5.3 Smart Block

SmartBlock [15] is a user-friendly blocks-based interface for the SmartThings IoT platform. They built their editor using Blockly [12], 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.

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

Plans for next semester

In the next part, I will focus on developing a user-friendly, blocks-based editor for IDP-Z3. In the next chapter, I will now elaborate on the reasoning behind this choice and outline the approach we plan to take for its development at this stage.

3.1 Blocks-based editor for IDP-Z3

In this section, we will discuss why we chose a blocks-based editor and how we see the implementation.

3.1.1 User-friendly interface

We are choosing to implement a blocks-based editor. The primary reason for this decision is the need to learn the, for non-experts, difficult FO(·)-syntax and the advantages that a blocks-based editor offers to address this challenges. The FO(·) language, along with its syntax, is not particularly readable for non-experts. In contrast, a blocks-based editor eliminates the need for syntax entirely, as it visually indicates what can and cannot be combined. This approach makes it impossible for users to make syntax errors.

Another significant benefit of a blocks-based editor is that it does not feel like traditional coding. Instead, it feels like the user is piecing together a puzzle that happens to produce code, which makes the process more intuitive and approachable. This is not always the case with other user-friendly editors. For example, while a CNL does not feel like coding either, its strict structure can make it feel unnatural, as discussed in Section 2.4.2 regarding the writability problem. A DMN is also visual, but lacks the expressiveness required for our modeling purposes. It is insufficient to describe only definitions when other useful symbols are needed but cannot be represented in DMN. Although cDMN resolves these limitations, it introduces additional complexity, making it less accessible for non-expert users.

Blocks-based editors have already proven to be highly user-friendly. A prominent example is

Scratch, which designed for children, who typically have no prior understanding of programming. Scratch's success highlights the effectiveness of blocks-based interfaces in simplifying programming concepts. Furthermore, as discussed in Section 2.5.3, existing literature frequently references blocks-based editors compared to other user-friendly alternatives. The study mentioned in that section, involving 33 participants, further supports the usability and effectiveness of blocks-based editors.

A crutual aspect of the editor we plan to make is validation of the rules created by the user. Others [14] have already proven that this is crutial for the user-friendlyness of such an editor. We plan to let the users interact with their ruleset through the Interactive Consultant of IDP-Z3, as this is already quite clear.

3.1.2 Structure and implementation

We intend to use Blockly, a blocks-based editor written in JavaScript. Blockly allows developers to create custom blocks with specific functionality. We will leverage this capability to design an interface that bridges the gap between a user-friendly editor and the FO($\cdot$) language. The editor will generate a ".idp" file, which can serve as a KB to be processed and reasoned over by IDP-Z3. We also aim to incorporate an interface similar to the Interactive Consultant discussed in Section 2.3.2. This addition will allow users to interact with their constructed KB, enabling them to test and verify whether the behavior aligns with their intentions. To ensure seamless communication and data transfer between systems, we will utilize Python.

3.1.3 Verification

We plan to conduct user testing with a group of individuals who have no background in home automation or programming. By asking them to model specific examples, we can effectively evaluate whether our goal of creating a user-friendly interface has been achieved.

3.1.4 Planning

My planning for the second semester is shown in Figure 3.1.

Figure 3.1: Planning for the second semester

Chapter 4

An FO(·) description for home automation

In this chapter we will discuss different ways to implement a home automation system in FO(·). The advantages and disadvantages of every implementation and what we will use going forward. To make it more understandable, we will use an example scenario shown in Figure 4.1.

4.1 Vocabulary

The vocabulary is an important part of the FO(·) description as it defines the possibilities and limits for the rest of the description. There are lots of different aproaches possible, which needs to be effective with the idea of ease of use and effectiveness in a real-world application. The first vocabulary we thought of is one that is easy to create from blocks due to the way devices are represented. Each device has its own constant, which is linked to an area. The states of these devices are represented through a type that defines all possible states for that device.

This vocabulary provides an easy and straightforward way to model a home; however, it does have its shortcomings. There is no way to address all devices (of the same type or not) at once, which is something the user would definitely want to do. Additionally, there is no way to create a structured representation of a house using the areas, which is not ideal.

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

     light1: → Area
     type stateOfLight <: Int
     stateOfLight1: → stateOfLight
 }
```

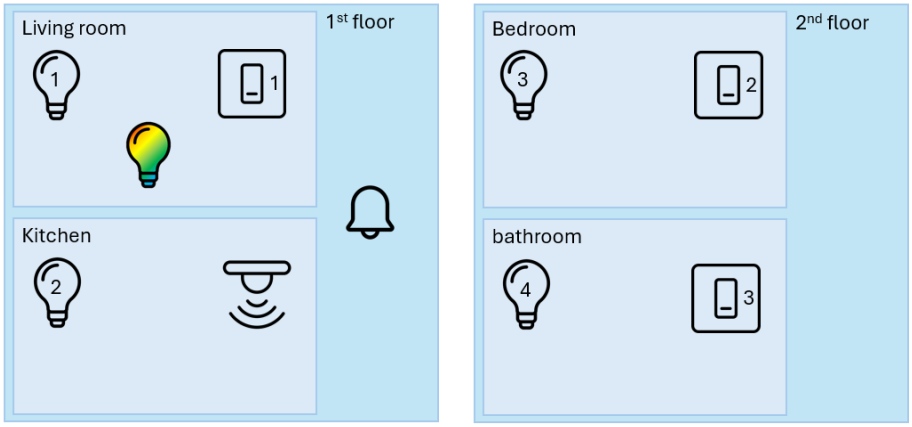

Figure 4.1: An example of a home automation system used to explain the different FO(·) implementations

The second vocabulary we explored improves on both of these shortcomings by defining a 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. The only 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.

```
vocabulary V {
    type Device
    type Area

    deviceIsInArea: Device → Area

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

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 sence that if all devices have different states, it will result in the same behaviour, but it opens up possibilities for more complex rules. This approach is slightly more challenging to construct using blocks, but the possibilities this vocabulary offers outweigh this drawback.

```
vocabulary V {
type Device
```

```
    type BinaryDevice <: Device
    type BinaryState := {on, off}

    binaryDeviceState: BinaryDevice → BinaryState
    }
```

Having experimented with all tree of the possibilities, we have decided to go with the third vocabulary. It offers the user the flexibility they want, while still using the IDP-Z3 reasoning engine to correct any mistakes that could be made.

An important note to make is the fact that these vocabularies limit the user in some way or another, this is by how the FO(·) language is desinged. However, since this is not just the FO(·) description, some functionality (that is not possible in FO(·)) can be implemented in the blockly backend. This is further explored in ....

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

[4] 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.

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] "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.

[14] 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.

[15] 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.