Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Dynamic Personalized Home Automation Rules based on Multi-User Feedback

Desislava Ivanova

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Brian Setz, M.Sc.
Commenced:	2020-11-02
Completed:	2021-05-03

Acknowledgements

I would like to thank the following people, without whose support this work would not have been possible:

- Prof. Dr. Marco Aiello for giving me the opportunity to do my Master's Thesis at the Institute for Architecture of Application Systems.
- M.Sc. Brian Setz for his consistent support, guidance, and comprehensive feedback.
- my partner Denis for proofreading and for his support, understanding, and encouragement.
- my parents for making my study possible and always supporting me.
- my friends and family for always being there when I need them.

Abstract

With the increasing use of smart devices in homes, the need for Home Automation Systems (HASs) has grown. A feature that most systems provide is automation rules. These rules are often defined in a format that is not easily understandable for all users and which may require prior knowledge or experience with HASs. Additionally, the rules are static, meaning that they need to be manually redefined by the user. This can be problematic since a single condition cannot be used in all scenarios. This can be improved by having dynamic rules that can be changed using user feedback.

The aim of this thesis is to examine how automation rules can be presented to the users of HASs in a human-readable form, and how automation rules can be dynamically updated after receiving explicit user feedback. Respectively, these two problems are also defined as the research questions of the thesis.

The first step of the research regarding the first research question is to come up with an approach that can translate an automation rule into human-readable text. For this purpose, existing research is examined. Afterwards, a solution is proposed and its implementation is thoroughly discussed. The proposed translation works with rules defined with the help of Node-RED. A triggered rule is parsed using a grammar created specially for the rules. The parsed rule is added to an Abstract Syntax Tree (AST) that is then used to form a comprehensible text that describes the essence of the rule.

For the purpose of answering the second research question regarding the dynamic actualization of automation rules, components that allow static rules to be dynamically changed using explicit user feedback are proposed. Additionally, a system is build with those components. The system uses existing components like Home Assistant (HA) and Node-RED and builds up on them, adding the components implementing the functionalities responsible for presenting the user with the rules, gathering his feedback, and updating the rules. Additionally, the translation approach proposed in the thesis is also included. The component responsible for the presentation of the rules and the collection of feedback is implemented as an Android application. The application presents the users with the text of a rule when it is triggered and gives them the ability to provide feedback on aspects of the rule. When the feedback is submitted it is used to update the corresponding rule in Node-RED.

The proposed system is then evaluated using different use cases. The use cases have different levels of complexity, meaning that they include a varying number of sensors and actuators. The evaluation of the system based on the mentioned use cases shows that the system is capable of translating automation rules into human-readable text and is able to dynamically update these rules with the feedback of the users. The evaluation also examines the limitations of the presented solution.

Kurzfassung

Mit der zunehmenden Verwendung von intelligenten Geräten im Haus ist der Bedarf an Hausautomatisierungssystemen gewachsen. Eine Funktion, die die meisten Systeme bieten, sind Automatisierungsregeln. Diese Regeln sind oft in einem Format definiert, das nicht für alle Benutzer leicht verständlich ist und das möglicherweise Vorkenntnisse oder Erfahrung mit Hausautomatisierungssystemen erfordert. Außerdem sind die Regeln statisch, was bedeutet, dass sie vom Benutzer manuell neu definiert werden müssen. Dies kann problematisch sein, da eine einzige Bedingung nicht in allen Szenarien verwendet werden kann. Dies kann durch dynamische Regeln verbessert werden, die mithilfe von Benutzer-Feedback geändert werden können.

Ziel dieser Arbeit ist es, zu untersuchen, wie Automatisierungsregeln den Benutzern von Hausautomationssystemen in einer menschenlesbaren Form präsentiert werden können und wie Automatisierungsregeln nach Erhalt von explizitem Benutzerfeedback dynamisch aktualisiert werden können. Dementsprechend sind diese beiden Probleme auch als die Forschungsfragen der Arbeit definiert.

Der erste Schritt der Forschung bezüglich der ersten Forschungsfrage besteht darin, einen Ansatz zu finden, der eine Automatisierungsregel in menschenlesbaren Text übersetzen kann. Zu diesem Zweck wird die bestehende Literatur untersucht. Anschließend wird eine Lösung vorgeschlagen und deren Umsetzung ausführlich diskutiert. Die vorgeschlagene Übersetzung funktioniert mit Regeln, die mit Hilfe von Node-RED definiert wurden. Eine ausgelöste Regel wird mithilfe einer speziell für die Regeln erstellten Grammatik geparst. Die geparste Regel wird einem abstrakten Syntaxbaum hinzugefügt, aus dem dann ein verständlicher Text gebildet wird, der das Wesentliche der Regel beschreibt.

Zur Beantwortung der zweiten Forschungsfrage bezüglich der dynamischen Aktualisierung von Automatisierungsregeln werden Komponenten vorgeschlagen, die es erlauben, statische Regeln durch explizites Benutzerfeedback dynamisch zu verändern. Zusätzlich wird ein System mit diesen Komponenten aufgebaut. Das System nutzt bestehende Komponenten wie Home Assistant und Node-RED und baut darauf auf, indem es die Komponenten hinzufügt, die die Funktionalitäten implementieren, die dafür verantwortlich sind, dem Benutzer die Regeln zu präsentieren, sein Feedback zu sammeln und die Regeln zu aktualisieren. Zusätzlich ist auch der in der Arbeit vorgeschlagene Übersetzungsansatz enthalten. Die Komponente, die für die Präsentation der Regeln und das Sammeln des Feedbacks verantwortlich ist, ist als Android-Anwendung implementiert. Die Anwendung präsentiert den Benutzern den Text einer Regel, wenn diese ausgelöst wird, und gibt ihnen die Möglichkeit, Feedback zu Aspekten der Regel zu geben. Wenn das Feedback abgegeben wird, wird es zur Aktualisierung der entsprechenden Regel in Node-RED verwendet.

Das vorgeschlagene System wird dann anhand verschiedener Anwendungsfälle evaluiert. Die Anwendungsfälle haben unterschiedliche Komplexitätsgrade, d. h. sie beinhalten eine unterschiedliche Anzahl von Sensoren und Aktuatoren. Die Evaluierung des Systems anhand der genannten Anwendungsfälle zeigt, dass das System in der Lage ist, Automatisierungsregeln in menschenlesbaren Text zu übersetzen und diese Regeln mit dem Feedback der Benutzer dynamisch zu aktualisieren. Die Evaluierung untersucht auch die Einschränkungen der vorgestellten Lösung.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Outline . . . 2

2 Background **5**
- 2.1 Home Automation Systems . . . 5
- 2.2 Home Assistant . . . 5
- 2.3 MQTT . . . 6
- 2.4 Node-RED . . . 7
- 2.5 Antlr . . . 8
- 2.6 InfluxDB . . . 8

3 Related Work **9**

4 Methodology **11**

5 Architecture and Design **13**
- 5.1 Architecture . . . 13
- 5.2 Rule Model . . . 15

6 From Node-RED to Translation Component **17**
- 6.1 Sending a Rule from Node-RED . . . 17
- 6.2 Receiving and Preparing a Rule in the Translation Component . . . 18

7 Conversion of Home Automation Rules in a Human-Readable Format **21**
- 7.1 Rule Grammar . . . 21
- 7.2 Parsing and Building an Abstract Syntax Tree . . . 25
- 7.3 From Abstract Syntax Tree to Text . . . 31

8 Collection of User Feedback **35**
- 8.1 Sending the Feedback to the User Interface . . . 35
- 8.2 Rule Location . . . 37
- 8.3 Feedback User Interface . . . 37
- 8.4 Correcting a Rule using the Feedback . . . 40

9 Evaluation **43**
- 9.1 Evaluation of the Translation . . . 43
- 9.2 Evaluation of Collection . . . 47
- 9.3 Evaluation of Research Questions . . . 48
- 9.4 Limitations . . . 49

10 Conclusion and Future Work **53**

Bibliography **55**

A Appendix **61**
A.1 Rule Grammar 61
A.2 Function Grammar 64
A.3 Use Case Rules in Node-RED JSON Format 66

List of Figures

2.1 Home Assistant's core system architecture. Used from [Homc]. 6
2.2 Message Queuing Telemetry Transport (MQTT)'s publish/subscribe architecture. Used from [MQTb]. 6
2.3 An Example Sequence of Nodes in Node-RED. 7

4.1 Research Methodology. 12

5.1 Node-RED Palette. Used from [Oped]. 14
5.2 Global Architecture of the System. 15

6.1 Example structure of a rule in Node-RED. 18

7.1 Example of a syntax diagram. 21
7.2 A rule as a syntax diagram. 22
7.3 A syntax diagram of ta. 22
7.4 A syntax diagram of $conditionTA$. 23
7.5 A syntax diagram of $eventsState$. 23
7.6 A syntax diagram of $stateCondition$. 24
7.7 A syntax diagram of $functionCondition$. 24
7.8 A syntax diagram of $serviceAction$. 25
7.9 A diagram depicting the rule Abstract Syntax Tree (AST). 26

8.1 The visual representation of the User Interface (UI) component. 38
8.2 Class diagram of the model for the rule in the UI component. 39

9.1 The first use case implemented in Node-RED. 43
9.2 The first use case's AST. 44
9.3 The second use case implemented in Node-RED. 45
9.4 The second use case's AST. 45
9.5 The third use case implemented in Node-RED. 46
9.6 The third use case's AST. 47
9.7 The rules of the use cases shown in the UI. 48

List of Listings

6.1 The function inside the "jsonFnk" node. 17
7.1 Example of simple JavaScript code that can be added to a function node in Node-RED. 24
8.1 Example structure of the JavaScript Object Notation (JSON) object sent to the UI. 36
8.2 Example structure of the JSON object sent to the Component for Updating Rules. 40
8.3 An example of the structure of the entries in the database. 41

List of Algorithms

6.1 Function that retrieves a rule sequence from an array of nodes. 19
6.2 Functions that reorder the nodes in a rule. 20
7.1 Function that finds the place for a state in the AST. 28
7.2 Function that finds a place for an action in the AST. 29
7.3 Function that finds a place for a flow node in the AST. 30
7.4 Function that starts the process of assembling the text of a rule. 32
7.5 Function for building a text from a block. 33

Acronyms

ANTLR ANother Tool for Language Recognition. 8

API Application Programming Interface. 7

AST Abstract Syntax Tree. v, xi, 2

GPS Global Positioning System. 50

HA Home Assistant. v, 5

HAS Home Automation System. v, 1

HTTP Hypertext Transfer Protocol. 17

IoT Internet of Things. 5

JSON JavaScript Object Notation. xiii, 7

MQTT Message Queuing Telemetry Transport. xi, 6

REST REpresentational State Transfer. 17

TA Trigger-Action. 15

TAP Trigger Action Programming. 15

UI User Interface. xi, 3

URL Uniform Resource Locator. 13

YAML YAML Ain't Markup Language. 5

1 Introduction

Smart devices or objects are becoming an important part of our everyday lives [RT17]. As a result, Home Automation Systems (HASs) have also become more popular in recent years [CP15]. They provide a large number of features. Nevertheless, to be able to use some of the functions, the users must have a specific expertise [BLM+11]. Automation rules are one of these features. An automation rule in a HAS determines what actions should be executed by certain conditions [New06]. In most cases, the ways in which automation rules can be defined have a learning curve, meaning that a random user cannot just start creating rules without being acquainted with the process first [BLM+11]. Studies show that regular users have problems with defining rules themselves [JOC+17][BLM+11]. They struggle with understanding the logic behind the rules or how they should be written. This can make users reluctant to use rules, as they increase the level of complexity of the system [BLM+11]. This can cause the users to not be able to benefit from the system to its full potential.

There can be two types of rules, dynamic and static. A static rule is created when the system is configured. If the user wants to change any of the parts of a static rule, he needs to manually redefine it. A study [BLM+11] found that it was not possible for users to use only static rules, as they cannot account for different circumstances. This can result in more challenges for users when they cannot redefine their rules on their own. This problem can be resolved by having dynamic rules. Dynamic rules are updated during the use of the HAS without the user having to explicitly redefine them. This is done by using user feedback. The feedback can be explicit or implicit. Implicit means that the system gets the feedback indirectly from the user by tracking his actions. The explicit feedback comes directly from the user.

1.1 Problem Statement

Most modern HASs allow the creation of automation rules, which are used to automate the user's environment. As those systems generally have multiple users, the rules need to correspond to the preferences of all users. Typically, the rules are pre-set. This can result in the rules being too strict for some users and too lenient for others. Another possibility is that they did not fit the actual preferences of the user from the beginning. There is a need to collect explicit user feedback after a rule has been triggered, as this provides the data needed to dynamically adjust the rules and correct them according to the user's feedback. In order for the users to provide accurate feedback, they need to be able to understand the automation rules. Furthermore, the way the feedback is collected and analyzed is also of importance, since it affects how precise the rules are for a given user.

The aim of this thesis is to research two main problems, namely the presentation of the automation rules to the users and the collection of users' feedback. First, this thesis examines how automation rules can be presented to users in a human-readable format. Additionally, it researches how the feedback of multiple users can be collected and used to dynamically update the automation rules in order to adjust the rules to the users' preferences.

The following research questions define the main objectives of this Master's Thesis:

RQ_1: **How can home automation rules be presented in a human-readable format?**
This question focuses on the form in which the user receives the automation rules that have been triggered. It aims to research how they can be translated in a format that can be understandable for all types of users.

RQ_2: **How can the users' feedback be collected after a rule has been triggered?**
This question aims to look into presenting the feedback to the user. Additionally, it looks into what happens with feedback once it has been submitted. It focuses on the collection, which is vital for making automation rules dynamic because with the gathering of the feedback comes also the updating of the rules.

1.2 Outline

This thesis is organized as follows:

Chapter 2 - Background provides the background for the topic of the research of this thesis. There, the concepts that are important for understanding the research conducted in this thesis are explained. Furthermore, the used technologies are presented.

Chapter 3 - Related Work explores other research close to the topic of this thesis. It looks into research about automation rules. Also, concepts investigating HASs that use feedback are presented. Furthermore, research that explores the translation of input into human-readable format is examined.

Chapter 4 - Methodology discusses the methods used for the research in this thesis. It describes the different steps of the research process.

Chapter 5 - Architecture and Design explains design decisions that were made to enable answering the research questions. It illustrates the architecture of the project and gives more information about the components that are used and will be added as parts of this thesis.

Chapter 6 - From Node-RED to Translation Component talks about the initial steps of the translation of automation rules. In this chapter, the connection between Node-RED and the translation component is discussed. Furthermore, it explains the preparations needed before a rule can be passed to the parser.

Chapter 7 - Conversion of Home Automation Rules in a Human-Readable Format presents the translation process of rules. The chapter is split into three sections explaining the big steps of the process. Section 7.1 introduces the grammar created for the parsing and its specifics. Section 7.2 talks about the actual parsing of the rules and how the Abstract Syntax Tree (AST) is build. The last section explains how a text is built from a parsed rule.

Chapter 8 - Collection of User Feedback deals with how the translated rules are presented to the user and how the user's feedback is then collected and used. It presents a User Interface (UI) concept that shows the rules and gives the users the possibility to add feedback about aspects of each rule. Then the chapter talks about how the feedback is used to update the rules.

Chapter 9 - Evaluation evaluates the research questions and discusses their answers. Furthermore, the concept for translation is evaluated using different use cases. The same is done with the method for collecting the feedback and updating the rules. The chapter also lists the limitations of the research done in this thesis.

Chapter 10 - Conclusion and Future Work summarizes and discusses the work and research done for this thesis. Afterwards, it presents topics that were part of the thesis which can be used as a base for future research.

2 Background

Home automation is a complicated topic which deals with many different concepts and technologies. The purpose of this chapter is to provide more information about the concepts and technologies relevant to the scope of this thesis. Section 2.1 introduces the topic of HASs. The next sections explain the technologies used to answer the research questions.

2.1 Home Automation Systems

A HAS comprises multiple smart devices and appliances that can be considered as separate entities operating in a shared environment [MPC+09]. The main parts of a HAS are its central controller, the UI, and the devices connected to it [GY16]. The role of the controller is to manage the devices connected to the system and communicate their states to the UI. The aim of a HAS is to make the users' activities in their homes more comfortable, economic, and secure.

2.2 Home Assistant

Home Assistant (HA) is an open-source HAS [Web19]. It is the largest system of its kind and has a giant community that supports it [GIT20]. HA provides over 1700 integrations [Home]. Both the system's core and its extensions are based on Python, which makes it easier to add new device support [Dag]. HA is easily configurable through its YAML Ain't Markup Language (YAML) configuration files. Figure 2.1 presents HA's core architecture. The core has the task of collecting information and controlling the connected devices [Homc]. As HA is an event-driven system, the Event Bus is its main part. It is responsible for triggering and listening for events. The State Machine manages the states of the connected devices. The Event Registry listens for services that have been called.

From HA's front-end, the user can see and control the state of the connected devices, manage automation and configure integrations [Homd]. The front-end is implemented as a progressive web application. The motivation for this is that the application needs to be mobile-first.

HA has a concept called Areas. Areas allow to group Internet of Things (IoT) objects together and add them to physical locations [Homb]. An Area does not provide information about its real location, meaning that it is not connected to the coordinates of the actual location. It is more of an organizing concept. Areas can be assigned to devices and not entities.

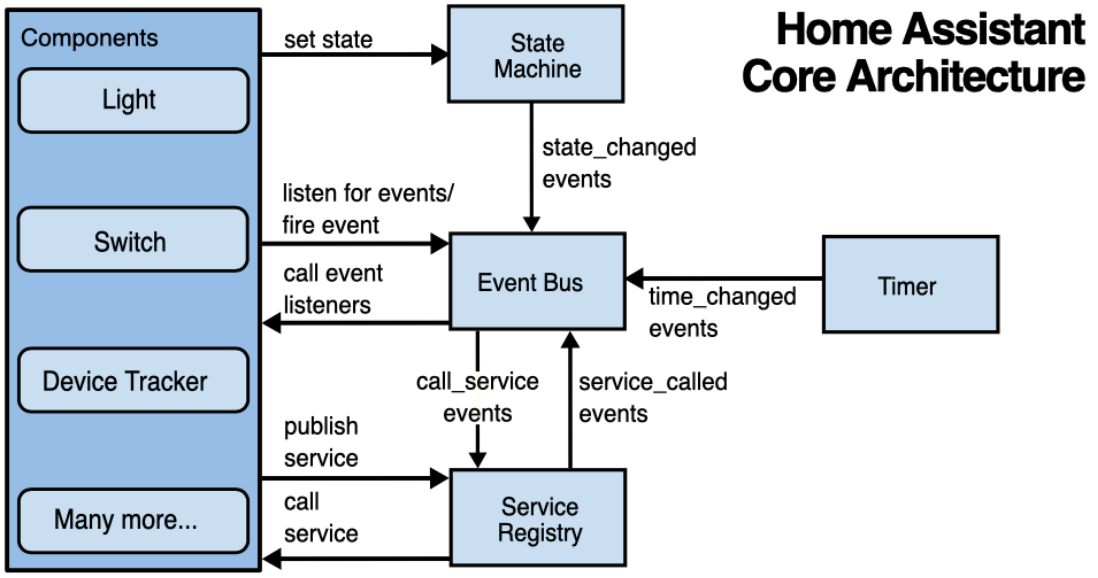

Figure 2.1: Home Assistant's core system architecture. Used from [Homc].

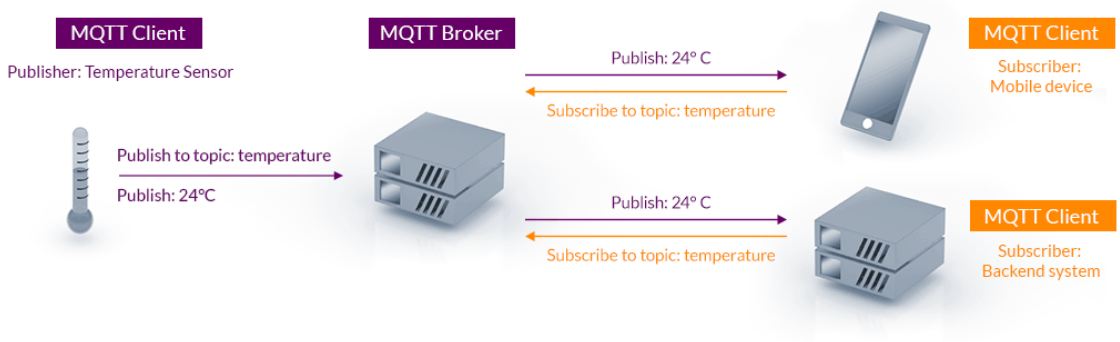

Figure 2.2: MQTT's publish/subscribe architecture. Used from [MQTb].

2.3 MQTT

Message Queuing Telemetry Transport (MQTT) is an OASIS and an ISO standard for communication between devices [MQTa]. It is a lightweight publish/subscribe messaging protocol. It has the following architecture: an MQTT client can publish messages on a specific topic to an MQTT broker [Nai17]. The destination of the message is determined by its topic [HTS08]. MQTT clients can subscribe to multiple topics and when there is new data published to those topics they receive it from the MQTT broker [Nai17]. Figure 2.2 shows MQTT's architecture.

2.3.1 Eclipse Mosquitto

Eclipse Mosquitto is an open-source MQTT broker [Eclb]. As Mosquitto is lightweight, it can be used on all types of devices, including low-power single-board computers. Eclipse Mosquitto's main task is to enable the communication between publishers and subscribers, playing the role of a communication channel [DS17].

2.3.2 Eclipse Paho

Eclipse Paho is an open-source EclipseIoT project that provides mainly client-side MQTT implementations [Ecla][DS17]. It has client libraries in many programming languages, such as Java and Python [DS17]. The Application Programming Interfaces (APIs) provided by the libraries can be used by both publishers and subscribers to connect to an MQTT broker.

2.4 Node-RED

Node-RED is a programming tool that allows users to link devices, services, and APIs together [Opeb]. It provides a browser-based editor where the user can easily create flows. The editor allows for the creation of JavaScript functions. Node-RED is based on Node.js. Because of Node.js' event-driven, non-blocking model, it can easily be run on low-cost hardware such as Raspberry Pi[1]. The flows created in Node-RED are stored in the JavaScript Object Notation (JSON) format, which makes them easy to import and export.

Flows are Node-RED's main organizing structure [Opea]. In the editor, they are represented by a tab. A flow can have a flow-scoped context that is only available to nodes in the specific flow. Nodes represent the elements that build a rule [Opec]. Each node has a specific function. Nodes are connected through their ports. Each node can have at most one input and multiple output ports. Figure 2.3 shows an example of a node sequence in Node-RED.

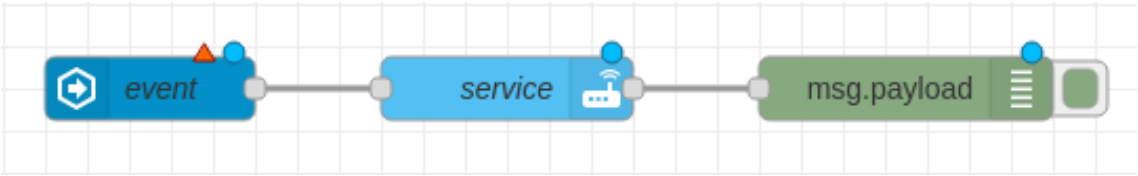

Figure 2.3: An Example Sequence of Nodes in Node-RED.

Nodes are connected to each other using wires [Opef]. A wire starts at the output port of a node and ends at the input port of another node. In the editor, wires are illustrated as the lines between the nodes. When the nodes are shown in the JSON format, each node has a wire list. There, for each output port, there is a list. In each list are the ids of the nodes whose input ports are connected to that output port.

[1] https://www.raspberrypi.org/

2.5 Antlr

ANother Tool for Language Recognition (ANTLR) is a language tool that enables the generation of parsers, which has been available since the 1990s [PQ95][Par13]. It is widely known and used in academic and in industry circles [Tera]. With ANTLR it is possible to create recognisers, compilers, and translators when provided a grammar [Par13]. When given a grammar ANTLR generates a Parser that can build and then traverse a parse tree [Terc]. ANTLR automatically generates a Listener and Visitor alongside the Parser and Lexer, which can be used to visit a specific node of the tree and execute some application-specific code at that point [Tera]. The latest version of ANTLR, ANTLR4, has libraries for a variety of programming languages including Java, Python, C++, and JavaScript [Terb].

Lexer: A Lexer is a software component that takes an input and breaks it down to vocabulary symbols that the Parser then uses [Par13].

Parser: A Parser is a software component that gives a grammatical structure to a given stream of vocabulary symbols [Par13].

Grammar: A Grammar describes a structure of a language in a formal way [Par13]. The Parser is created based on the grammar so it can recognize the language the grammar describes.

2.6 InfluxDB

InfluxDB is an open-source time series database [Infb]. A time series database is intended for time-stamped or time series data [Infd]. Time series data is data that includes values that are measured over a period of time. The database is mainly used in the fields of IoT, monitoring, and analytics [Infb]. This results from the fact that it can manage large amounts of data that are time-stamped. The entries in the database follow a specific format. It has four types of components [Infc]. These are measurement, tag set, field set, and timestamp. The measurement holds the name of the measurement that is saved. The tag set contains tags that record metadata [Infa]. They are indexed and can be used in queries. The field set holds the fields which record metadata and data value. Because they are not indexed, they cannot be used in queries. The timestamp records the time of the measurement.

3 Related Work

As automation rules are an important part of a HAS, they have been discussed in many scientific works. Drey et al. [DMC09] discuss rules, but only as a vital part of a HAS. In [NLC02] the importance of rules is also stated, taking them into account when describing their internet-based control architecture for HASs. This thesis also discussed home automation rules, but unlike the papers that were just mentioned, it goes into much more detail and researches their structure.

[NE16] and [BXL+18] both look into the topic of improving rules. The first paper proposes a tool that can identify if a rule has been defined correctly, checking if the triggers for the rules are enough for the desired tasks. Similar to this thesis, it researches the structure of rules and how they should be defined. Nevertheless, it does not look into dynamic home automation rules as this thesis does. The second paper shows a platform that can fix rules so they satisfy the expectations of the user. Similar to our work, the paper focuses on home automation rules and their importance to the user. Nonetheless, as the previous papers, it does not research the field of dynamic home automation rules. Dynamic rules are discussed by Brush et al. in [BLM+11], but there, they only state the need for them. Our research looks into how dynamic rules could be changed with the help of the user's feedback.

There are many studies where the effects of a HAS giving feedback to its users are researched (e.g. [BHSB17] [KT14]). Only a few have looked into the opposite - the users giving their feedback for the actions of the system. Ham et al. [HSK13] used user feedback together with a Learning-Based Model Predictive Controller, so it could create a home model for an air conditioning system that is dictated by the comfort of the user. Similar to the system discussed in this thesis, the system they provide is feedback-driven. The automation rules, however, are not shown to the users, as in our work. Their system only studies the users' desired daily temperature patterns. The possible feedback is also only related to temperature, checking if it is too warm or too cold. In this thesis, the rules that are examined are more generalized and can include various types of sensors and actuators.

Rashidi et al. [RC09] introduce the CASAS system. It gathers information about the users' activities and automatically creates rules using machine learning techniques. When the rules are created, it takes into account the users' explicit or implicit feedback so it can dynamically update its model. In our work, we also look into updating the users' defined rules in accordance with their feedback. The CASA-U, the system's UI, is used for gathering the users' feedback. Unlike our system, where we look into presenting the rules in a human-readable format, here the rules are presented to the users as an animation. Also, the feedback that the users can provide only includes how much they liked the whole rule and not specific aspects of it. The aspects can be changed, as well, but this needs to be done manually. In our proposed system the users can give feedback to specific characteristics of a rule.

Another architecture that can dynamically adapt automation rules in accordance with the users' feedback and behavior is provided by Karami et al. [KFBL16]. As the system of Rashidi et al. [RC09], it first uses activity recognition, which looks for rules that can be created from the users'

behavior. Similar to this thesis, the users' feedback is of great importance. In conformity with the users' implicit or explicit feedback, the found rules can be changed to potential preferences, which can be used to update the automation rules. When a rule is triggered, the only feedback that the users can provide is if they like or dislike the whole rule - using thumbs up or thumbs down. This is different from our system, where we want to allow changes of specific aspects of a rule. In the case of rules that include multiple triggers, the given approach cannot supply enough information about the trigger conditions, whereas our approach works even with rules that are more complex. Moreover, when a rule is triggered, it is not shown, the users have to be in the same physical space to comprehend that something has happened. In our proposed system the rule is shown in a human-readable format.

Khan et al. [KSZ17] propose a framework that provides user-centered assistance in getting an overview of a HAS's functionalities and optimizing its automation rules. The available functions are shown on the Adaptive User Interface, where the rules can also be updated. In our research, we also explore how home automation rules can be updated. Another component of the architecture called OLAC is responsible for the optimization of the automation rules. First, implicit feedback is used to determine users' behaviors that could change existing rules. Then, the users are asked for explicit feedback so that the rule can be changed. Unlike our work, Khan et al. do not discuss how exactly the rules are presented to the users. Additionally, asking for explicit feedback only after implicit feedback has been gathered does not let the users express their opinion for the rule but only for the system's suggestion. In this thesis the user feedback is concerning the automation rules.

Tsuchiya et al. [TYA18] discussed a method for converting complex chemical structures to a human-readable format. As in this thesis, they used a parser to translate the data structure representing the chemical structure into another format which is human-readable. The difference between the method of Tsuchiya et al. and this thesis is that they convert one existing format into another existing format that is human-readable, whereas here, we will convert an existing format (the rule) into a text that describes what has happened in the rule in question.

No works were found that talk about showing users home automation rules in a human-readable format. In our work, similarly to the ones listed here, we will look into human-centered home automation. We, however, will focus on explicit user input. Instead of researching how well behavioral patterns have been detected, the feedback will be used for updating the present automation rules. Furthermore, we will research how the rules could be shown to the user in a human-readable way.

4 Methodology

A home automation setup usually is made of a HAS that controls and manages the devices connected to it. If the setup supports automation rules, it should also have a component that deals with this task. Without such a setup, it is not possible to use home automation rules. To be able to answer the research questions, setting up the base components will be the first task for this work. Afterwards, the structure of home automation rules will be researched in order to determine a rule model that can be used later on.

The next step will be to create a translation component that translates a rule into human-readable text. Parsing is a technique used to determine the syntax of given inputs [Cha87]. Therefore a parser is selected as the basis of the translation component. A context-free grammar for the parser has to be created. It needs to be based on the rule syntax and the determined rule model to be able to recognize the input. The parsed rule will then be put in a parse tree. An AST will be built from the parse tree. An AST will be utilized for this since it can directly be used to build a text. The parse tree will include the whole syntax of the input, whereas the AST will contain only the elements that would be part of the textual representation of the rule. The AST will then be traversed, and a text will be generated from it. To ensure that the rule syntax always complies with the structure of the context-free grammar, a reordering algorithm will be created.

In order to present the text to the user and to collect his feedback, an Android application will be created. Every time a rule is triggered, the application will show the rule's text to the user. Afterwards, the feedback will be collected and send to another component. This component's role will be to calculate the values that will be used to update the rules. As the system will support multiple users, it has to be able to change a rule when having several feedbacks for it.

To be able to the evaluate the rules, the following use cases are defined:

- One sensor (temperature sensor) and one actuator (electric fan): The fan is turned on if the temperature is higher than a given value. If the temperature is lower than the same value, the fan is turned off.
- Two sensors (motion sensor, light sensor) and one actuator (lamp switch): The light is turned on when the motion sensor detects movement, and the light sensor detects a value under a certain given value.
- Four sensors (temperature sensor, motion sensor, window sensor, light sensor) and three actuators (HVAC system switch, window actuator, blinds actuator): When the motion sensor detects movement and the window is open, and the temperature inside is above a given value, then the window should be closed, and the HVAC system should be turned on. Afterwards, the value of a light sensor is checked, and if it is higher than a specific value, then the blinds are closed.

The use cases vary in their complexity, the first being the least complex, only having one trigger and one action, and the last being the most complex, having four triggers and three actions.

Flow Chart 4.1 shows a graphical representation of the workflow of this thesis and the order in which the steps were completed.

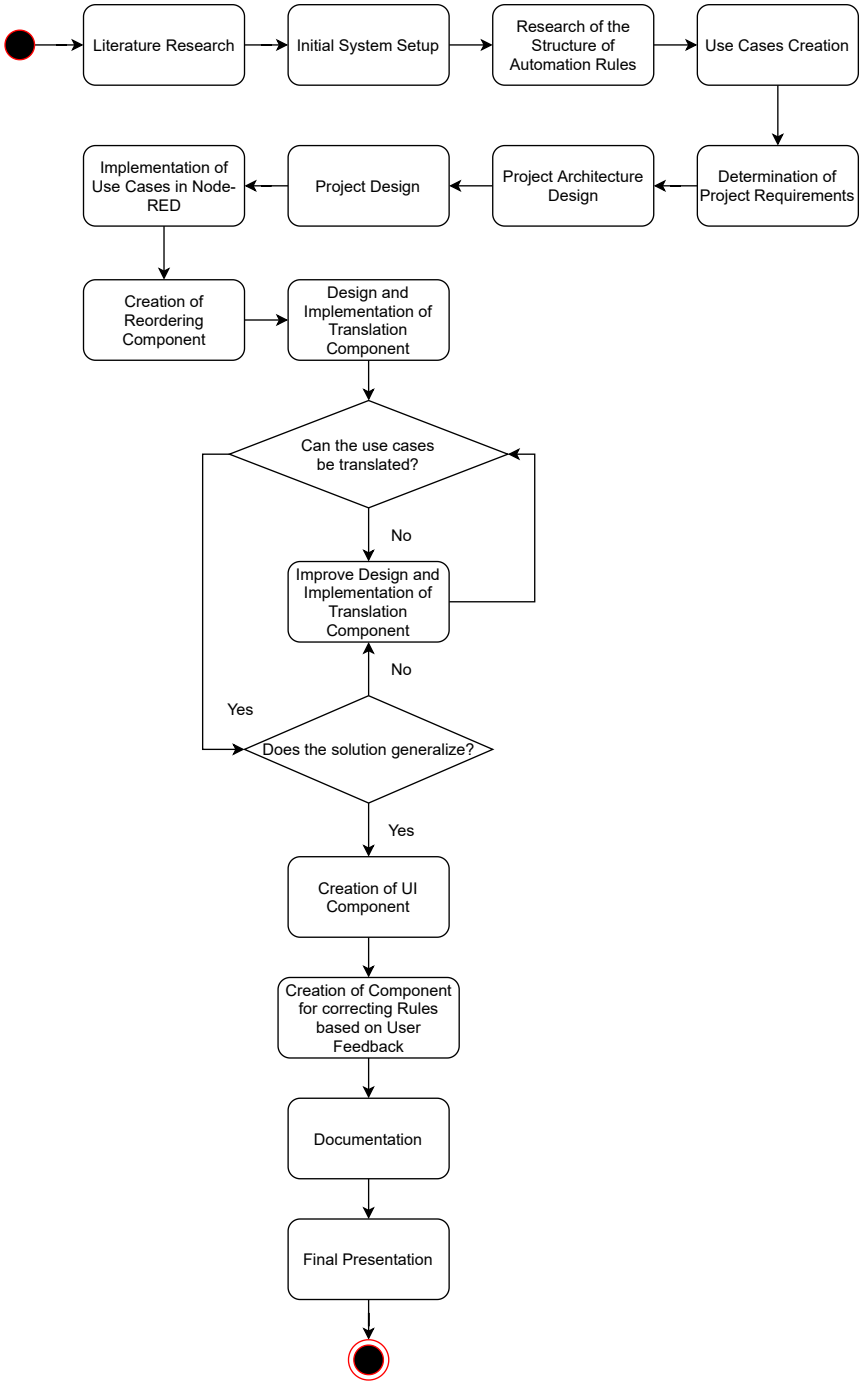

Figure 4.1: Research Methodology.

5 Architecture and Design

This section discusses the architecture of the system used in this thesis and the design decisions behind the technologies that were chosen. This is presented in Section 5.1. Subsequently, in Section 5.2 the rule model that is used in the later chapters is explained.

5.1 Architecture

As mentioned in Chapter 4 the base setup of the system should consist of a HAS and an automation rule engine. The HAS manages the connections between the entities that are part of the system. HA was chosen as the HAS because it is popular and is one of the best currently available open-source systems [GIT20]. Node-RED, a rule engine, is also added to the setup because it is also well known and provides more flexibility because it can be used with other HASs. Both systems and their characteristics are described in Chapter 2. Docker is used for the setup of the existing systems.

Node-RED is connected to HA by adding HA nodes to Node-RED's palette. The palette in Node-RED contains all of the various types of nodes that can be used in the Node-RED flows [Oped]. The nodes in the palette are ordered in categories, making it easy for the users to understand their functions. The palette is shown left of the flows in the editor. Figure 5.1 shows how it looks. New nodes can be added using Node-RED's Palette Manager [Opee]. For the connection a module called "node-red-contrib-home-assistant-websocket"[1] is used. It allows to integrate Node-RED with HA using HA's base Uniform Resource Locator (URL) and access token. Once the connection is configured on Node-RED's side, it can be used for every node from the module.

Figure 5.2 shows the global architecture of the system. The components shown in red are the ones that had to be implemented for this thesis. Due to the SARS-CoV-2 restrictions, the access to the university was restricted, so it was not possible to use the sensors and actuators provided by the university. Because of this, they had to be mocked.

The connection between the mock sensors and HA is made using the MQTT messaging protocol. An MQTT broker implementation called Eclipse Mosquitto is used to establish the connection. It is set up using Docker. To connect HA to it, the broker's address had to be added to HA's configuration file. MQTT is used since HA has integrations for different types of MQTT sensors and actuators. The sensors and actuators are implemented using Eclipse Paho's MQTT client implementations. The sensors' and actuators' characteristics had to be added to HA's configuration file to be accessible from HA.

[1] https://flows.nodered.org/node/node-red-contrib-home-assistant-websocket

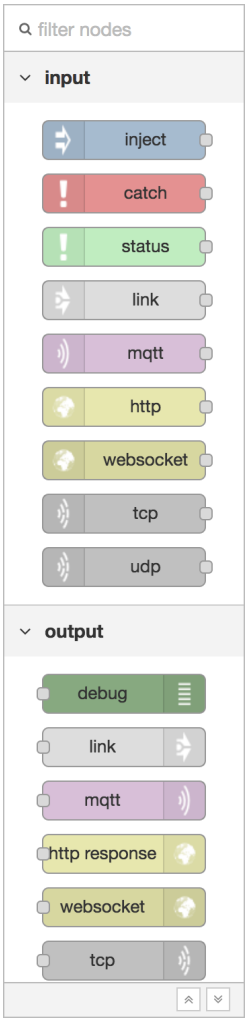

Figure 5.1: Node-RED Palette. Used from [Oped].

HA provides Node-RED with the current state of the sensors that are connected to it. If at some point a rule in Node-RED is triggered, the actions of the rule are sent back to HA. HA then sends the commands back to the actuators through the MQTT broker.

The features that are implemented in order to answer the research questions are split into three components. The first is the translation component. It receives a triggered rule from Node-RED. First, the rule is prepared for the parser. The rule model presented in Section 5.2 is also used in the preparation and parsing of a rule. The exact steps of the preparation are discussed in Chapter 6. The rule is then parsed and translated to a human-readable text. The exact process will be discussed in Chapter 7.

After the rule is translated, it is sent to the UI. The connection between those two components is also established using an MQTT broker. The UI is implemented as an Android application. A stationary implementation like a normal computer application would not be a practical solution since the users would be required to always be around their computers. Hence a mobile application that can show the rules to the users regardless of their location in the home is a more suitable solution. When

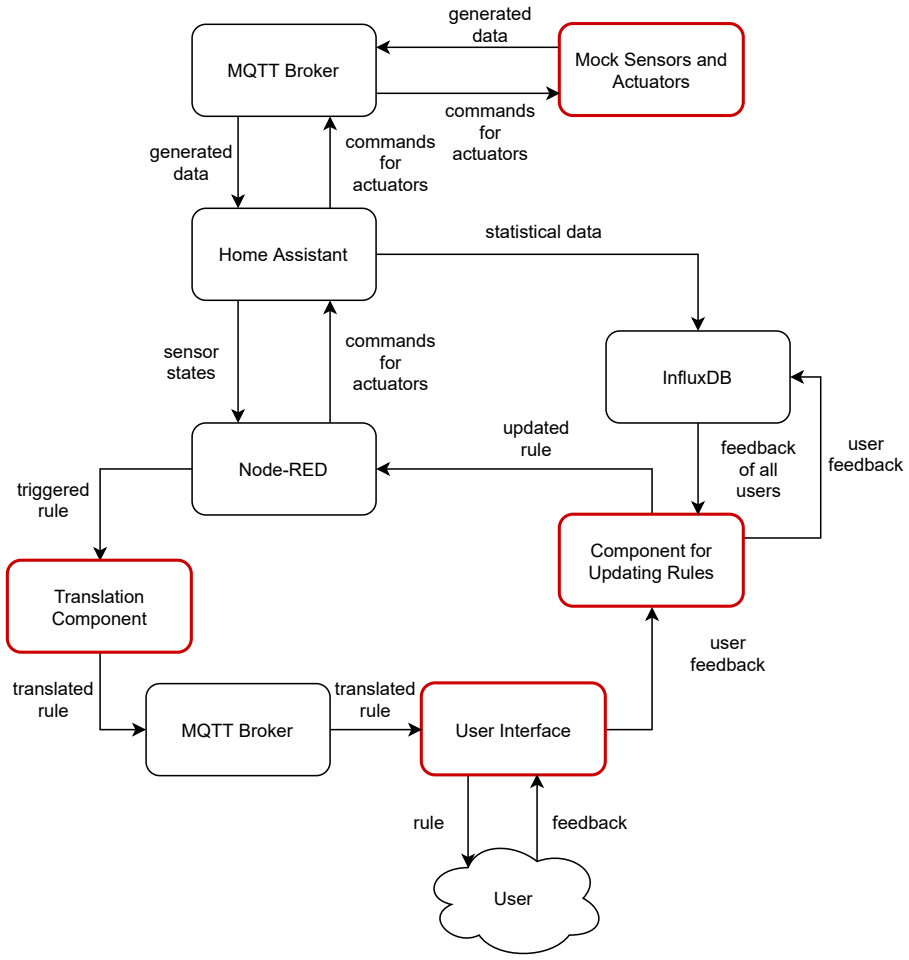

Figure 5.2: Global Architecture of the System.

the users are notified that a rule has been triggered, they can give their feedback for the rule. The feedback is then sent to the final component of the system. This last component takes the users' feedback and stores it in an InfluxDB database. From the database, it retrieves the feedback of all users for the given rule and calculates the mean values for the rule. The changes to the rule that has been reviewed are then applied. Those two components are examined more thoroughly in Chapter 8. Finally, if changes are made to the rule, the updated rule is sent back to Node-RED so it can be updated before it is triggered again.

5.2 Rule Model

To be able to translate home automation rules into coherent text, the rules need to have a structure. For this purpose, a rule model needs to be determined. Dey et al. [DSSK06] made a study researching context-aware applications. The results indicate that the most common mental model for a rule is if-then. This mental model is used in the concept of Trigger Action Programming (TAP). The Trigger-Action (TA) characterizing the if-then concept. In a TA pair a trigger is defined

by an event and a condition [GMPS17]. In the context of home automation, an event can be defined as receiving a value of a sensor. The condition determines by what value an action should happen. An example of a trigger is "when the value of a temperature sensor is above 20 °C". The action determines what happens when the condition is fulfilled. It can be something like "open window". As there can be multiple cases in an if-clause, there can be multiple triggers. The triggers can be followed by multiple actions. TAP allows users to define the behaviour of a system using the TA model.

TAP is used in many systems for adding automation rules [UMPL14][BV15]. An example are HASs like HA and openHAB[2] which allow to directly add rules using their UI [Homa] [ope]. Another example is the tool IFTTT[3]. It is a tool that can be connected to HASs. There, the user can define the rules by adding triggers and actions to the sentence "If this(trigger) that that(action)". There are also applications that use different methods for adding automation rules. Examples are formula languages and/or visual programming [BV15], but because of their higher complexity, they are not substantial for building a text. Therefore, they are not relevant to this thesis.

Ur et al. [UMPL14] conducted a study that shows that the average user's most desired behaviors from a home automation system can be expressed through the TA model. Their study shows that 78 % of their participants used only one trigger and one action. 22% of the study's participants needed to use either more triggers or more actions. Because of its simplicity, the TA model is chosen as a base for the rule model that is used in this work. Nevertheless, as Ur et al.'s research shows, there is a percentage of users whose needs are not fully supported by the simple model. Therefore in this thesis, there are slight extensions to the model which can support them.

In this thesis, the TA model is used for the rules. Here we will call each TA pair a TA component. A rule is not build out of a single TA component. It can have multiple concatenated TA components. A simple example for such more complex rule is:

- If the temperature is above some value, turn the air conditioning on. After that, if the light outside is too bright, close the blinds.

As can be seen, such a rule has two TA components. Furthermore, if the TA component is considered as an if-clause, then the rule model used in this thesis can also support else-if- and else-clauses. An example for this would be the following rule:

- If the temperature is above some value, turn the air conditioning on. If the temperature is equal to or below that value, turn the air conditioning off.

The example shows a rule made out of an if- and an else-clause. The model is discussed further in Chapter 7.

[2] https://www.openhab.org/
[3] https://ifttt.com/

6 From Node-RED to Translation Component

This chapter deals with the preparation of rules before they can be parsed. In Section 6.1 the preparation from Node-RED's side is explained. There the steps that are taken so that a rule defined in Node-RED can be sent to the translation component are discussed. Section 6.2 talks about the processing of the rules when they are received by the translation component. There, the construction and order of the rules are explained.

6.1 Sending a Rule from Node-RED

For popular systems like HA, Node-RED already has specific integrations which allow the transfer of data between the systems. As in this thesis, data needs to be sent from Node-RED to the translation component, the systems should be connected in such a way that allows for sending all of the information about a rule. Since there is no ready solution for this available, a custom method for sending automation rules to the translation component needs to be determined.

The connection between Node-RED and the translation component is established using REpresentational State Transfer (REST). Node-RED provides a specific node type, which sends Hypertext Transfer Protocol (HTTP) requests. When a rule is created in Node-RED, the last node should be an HTTP request node, which sends a request to the translation component. Because the message that is transferred between the rule nodes includes only information about the last state of a entity that has been looked up, another node is added between the last node, which is part of the rule logic, and the HTTP node. This node is always named "jsonFnk", so it can be distinguished from the

Listing 6.1 The function inside the "jsonFnk" node.

```
var list = flow.get("list_payload");
var ids = ["2c8d3ef7.2ebc02","2ac72721.cb3328","e737ca83.49fb9"];

if (list == null){
    list = msg.payload;
}

msg.payload = {
    "entity_id":null,
    "payload": list,
    "ids":ids};

flow.set("list_payload", null);

return msg;
```

other function nodes used in the rule. Listing 6.1 shows the function that this node has. It sets the payload of a message to a specific format.

The format is a simple JSON object(lines 8 to 11) that includes the ids of the start nodes of the rule(line 11) and information about the sensors and their last states(line 10). If the rule has only one sensor, then its entity id is added to line 9, and only its payload is added to line 10. As there is no way to automatically get the ids of the start nodes, they have to be manually added for each rule, as shown on line 2. If there are multiple sensors that are sequentially spread through the rule, the payload of the last node will not contain all of their values. Because of this, in such cases, the values of sensors are added to a list that is saved as a flow context variable. The if-clause on line 4 checks if this is the case. If it is not, the normal payload of the message is used. When the JSON is filled, the list variable is emptied so it can be used again the next time the rule is triggered.

Figure 6.1 shows how a very simple rule looks. The last two nodes are the ones that were just discussed. They need to be part of any rule in the exact order they are seen in. Without them, it would not be possible to continue with the translation process.

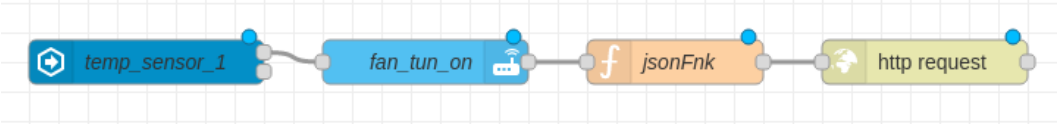

Figure 6.1: Example structure of a rule in Node-RED.

6.2 Receiving and Preparing a Rule in the Translation Component

When a rule is triggered, it is sent to the translation component. There, the data received from Node-RED cannot be directly parsed because it does not contain a whole rule but only information about its start nodes. It is not possible to make a Node-RED rule send more information about the nodes without having to have specific hard-coded information for every rule. Therefore, the rule first needs to be constructed using the information received in the request, before it can be used.

Node-RED's HTTP request is received by a simple HTTP server. The server and the whole translation component are written in Python. The contents of the request, as shown in the last section, only contain the ids of the start nodes of the rule. Node-RED has an API that provides all of the created flows. The flows are supplied as a list of JSON objects representing the nodes. All JSON objects have a node id and a list of wires. Using the ids provided in the HTTP request, the start nodes' JSON objects can be determined. From there, using their wires, the entire sequence of nodes can be traversed.

Algorithm 6.1 shows how a rule is assembled from the JSON list. First, the start nodes are retrieved from the list provided by Node-RED(lines 5-12). They are added to a list that represents the rule that is of interest(the "rule" list). Each start node's wires(the ids of the next nodes in the rule) are put in another list which represents the next nodes that need to be added to the rule list (the "next" list). After all start nodes have been appended to the "rule" list, one by one, the next nodes are retrieved(lines 13-32). They are taken from the list the same way the start nodes were, by their ids. When a node is found, its wires are added to the "next" list if they are not already in the list or in the

"remove" list(lines 21-24). The "remove" list is used to track wires whose nodes have already been added to the "rule" list. This is repeated until there are no more wires in the "next" list. The check on lines 17-19 is used to make sure that the nodes that are not part of the rule logic(e.g "jsonFnk" node) are not added to the "rule" list.

Algorithm 6.1 Function that retrieves a rule sequence from an array of nodes.

```
1:  procedure GET_RULE(nodes, ids)// nodes = a list of JSON objects representing nodes, ids = a
    list of the ids of the first nodes in a node sequence
2:      initialize rule = []
3:      initialize next = []
4:      initialize removed = []
5:      for node in nodes do
6:          for id in ids do
7:              if node[id] is equal to id then
8:                  add node to rule
9:                  add all wires of node to next
10:             end if
11:         end for
12:     end for
13:     while next is not empty do
14:         for node in nodes do
15:             for wire in next do
16:                 if node[id] is equal to wire then
17:                     if node is a helper JSON format node then
18:                         add wire to removed
19:                         remove wire from next
20:                     else
21:                         add node to rule
22:                         add wire to removed
23:                         remove wire from next
24:                         add all wires of node to next if they are not already there or in removed
25:                     end if
26:                 end if
27:             end for
28:             if next is empty then
29:                 break for loop
30:             end if
31:         end for
32:     end while
33:     return rule
34: end procedure
```

To be able to parse a rule, the nodes in the list that Algorithm 6.1 returns need to be ordered in a specific way that complies with the grammar of the parser. If the nodes are not ordered in the way defined by the grammar, the parser is not able to recognize the rule, and it cannot be parsed. The desired order is based on the rule model discussed in the previous chapter. A rule can be built out of multiple TA components. To be able to get the right meaning of the different components, they

should not be mixed with one another. Therefore, in the order of the rule, all of the nodes building one TA component should be added to the rule before the nodes of the next one are added. When a rule is retrieved by Algorithm 6.1 the nodes that build the different TA components are mixed up. Because of this, the nodes in the "rule" list that is returned by the algorithm need to be reordered.

To ensure this, the following algorithm was created to reorder the list. Algorithm 6.2 shows how the nodes are reordered. The idea is that after a condition, there could be multiple actions before there is another state condition(node types in Node-RED that build a trigger). Because the rule graph is traversed in a breadth-first manner, a state(event) or a condition might be added to the rule list before all actions of the previous were. To assure the right order is kept, each node is given to a procedure(ordered()) that checks if its successor is an action node(line 5). If that is the case, that action is added to the list before the node that was at that position in the initial order(line 16). Since it is possible to have a chain of multiple actions, the successors of the added action are checked recursively(line 17) until the next node is not an action(lines 10 and 22).

Algorithm 6.2 Functions that reorder the nodes in a rule.

```
1:  procedure FIX_ORDER(rule, ids)    // rule = list of nodes that form a rule, ids = the ids of the
    nodes in the rule list
2:      correct_order = []
3:      for node in rule do
4:          add node to correct_order
5:          correct_order = orderer(rule, ids, correct_order, node)
6:      end for
7:      return correct_order
8:  end procedure
9:  procedure ORDERED(rule, ids, co, node)// rule = a list of nodes part of a rule, ids = the ids of
    the nodes in the rule list, co = a list of nodes of a rule with the correct order they should be in,
    node a node
10:     if node[type] is not one of the state or condition types then
11:         for wire in node[wires] do
12:             if wire in ids then
13:                 get index i of wire in ids
14:                 get node next_node from rule using i
15:                 if next_node[type] is not any of the state of function types then
16:                     add next_node to co
17:                     co = orderer(rule, ids, co, next_node)
18:                 end if
19:             end if
20:         end for
21:     end if
22:     return co
23: end procedure
```

7 Conversion of Home Automation Rules in a Human-Readable Format

This chapter presents the approach for translating home automation rules into a human-readable format. First, in Section 7.1, the grammar for the parser is introduced. There, the specifics of the grammar and its rules are discussed in detail. This grammar is used by the parser to build a parse tree. After the tree has been built, it is traversed, and parts of it are used to build an AST. This is discussed in Section 7.2. In Section 7.3, the way the AST is walked so that a text can be build from it is explained. The formation of the text is described in the same section.

7.1 Rule Grammar

The grammar that is used for the parsing of rules is based on the TA rule model and on Node-RED's node syntax. As already mentioned, Node-RED provides its rules in a JSON format. Consequently, the grammar also has to support this aspect of the syntax.

A formal grammar is defined by using terminal and non-terminal symbols [GJ91]. Terminal symbols are real symbols like words or punctuation marks. Non-terminals are an abstraction that is used to combine terminals and non-terminals together. If an analogy between a grammar and a tree is made, then the non-terminal symbols are the nodes of the tree and the terminal symbols are the leaves.

The parts of the grammar shown in this chapter are presented using syntax(or railroad) diagrams. In the diagrams shown here, non-terminal symbols are represented as boxes with rounded corners, whereas terminals are normal black boxes with blue text inside. An example for this is shown on Figure 7.1. The highest line after the terminal symbol indicates that there need not be any other symbols after it. The line under it shows that if there are, there can be multiple of them. The way the two non-terminals are placed shows that there are two possibilities.

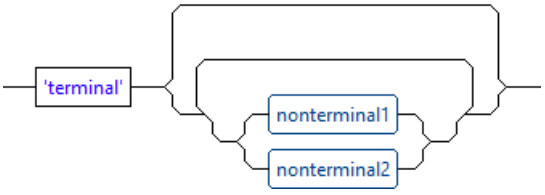

Figure 7.1: Example of a syntax diagram.

Figure 7.2 shows the structure of a rule. A rule must have at least one TA component, here represented by the *ta* non-terminal. As previously discussed, a rule can contain multiple TA components. Here this is represented by the second half of the diagram. After the first *ta*, there can be either another *ta* or a *conditionTA*. Both represent TA components, although there are some differences in their structure, which are discussed in the following paragraphs.

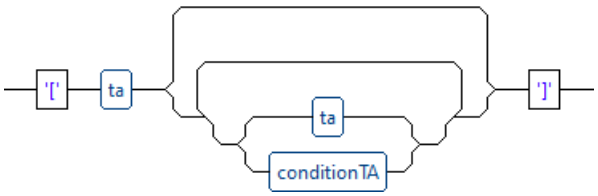

Figure 7.2: A rule as a syntax diagram.

An analysis of Node-RED's nodes showed that for the purpose of automation rules, the nodes could be separated into four types - state, condition, action, and flow nodes. State nodes are nodes that retrieve the state of a HA object. These objects are mainly sensors. Conditions are nodes that require a condition to be passed in order for the information flow(messages) to continue to the next node. Actions are nodes that trigger some specific action, e.g. trigger some behavior in a HA actuator. Flow nodes are nodes that influence how the information flow(messages) is transferred or formatted. An example would be joining the paths of multiple nodes and outputting a single path afterwards.

As previously discussed, in a TA component, a trigger is defined by an event and a condition. For Node-RED to determine that something has happened in HA, it needs to receive the data from the HAS. This is the job of the state nodes. They receive data from HA's entities. This means that the state nodes are the event part of a trigger. Therefore in the grammar, a *ta* non-terminal starts with a state node. Figure 7.3 shows the structure of *ta*. It can have multiple states and afterwards a condition. The states, together with the condition, play the role of a trigger(or triggers). There may be no condition node because, in some of Node-RED's state nodes, the condition is part of the node. After the condition, there is at least one action node. In this order, those three node types build a TA component.

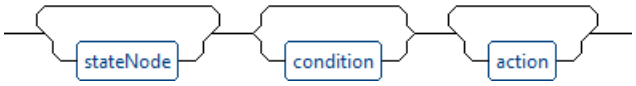

Figure 7.3: A syntax diagram of *ta*.

After the first *ta* there can be multiple *ta* or *conditionTA* non-terminals. A *conditionTA* also represents a TA component. The difference between it and the *ta* is their structure. As can be seen in Figure 7.4 a *conditionTA* does not contain a state. It is possible to have a TA component without having a state node in it as there is at least one state in the previous TA component in the rule. That TA component's state's value can be used in the condition of the *conditionTA* to create a trigger. After the condition, there is at least one action, similar to the *ta*.

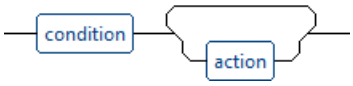

Figure 7.4: A syntax diagram of *conditionTA*.

In this grammar, *stateNode*, *condition*, and *action* non-terminals are responsible for the whole structure around the nodes that they contain. This means that they include a non-terminal that generalizes all of the nodes of their node type and parts of the JSON syntax. A *stateNode* includes a *stateNodeType*, and if there are multiple state nodes, it also includes the commas between them. The *condition* non-terminals are similar, but they always include a comma. The difference is that a *condition* can include a *flowNode* as well as a *conditionNodeType*. The *action* non-terminal only includes an *actionNodeType* and a comma. They are not shown on any of the figures exhibited here but can be found in Appendix A.1, where the whole grammar is presented.

The *stateNodeType* represents each of the state node types included in Node-RED. As discussed in the previous paragraphs, the state node represents the event part of a trigger. Each state type has a different syntax, although they are quite similar. It is possible for state nodes to have an inner condition. In cases where there is just one state, this inner condition can be used instead of a normal condition to trigger a rule. There the trigger of the TA component is represented by that one node. Figure 7.5 shows the syntax diagram of the *eventsState* type. It follows the syntax of the JSON node. The parts of the state type that are of importance have a specific syntax of their own, e.g *nodeId*. The type non-terminals are used to distinguish between the node types. Each different node type has a unique type key/value pair. For the *eventsState* it is the non-terminal called *stateType*. The non-terminal *entityId* represents the id of the HA entity that is connected to the node. Parts of the node that are not significant for the translation are generalized as *dump*. As the *dump* contains irrelevant information, it is structured as a set of standard JSON key/value pairs without any specific meaning. The *wires* non-terminal contains the ids of all the nodes which come directly after this node. The *stateCondition* incorporates the inner condition of this type of state node. If *stateNodeType* has a valid inner condition, then they represent the trigger part of the TA component and there is no need for a *conditionNodeType*.

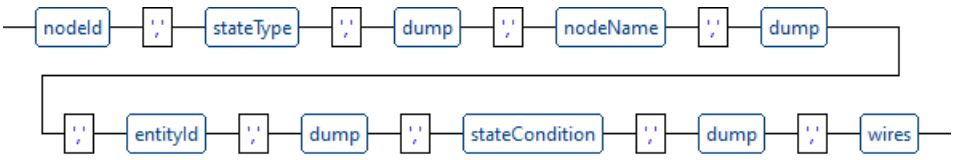

Figure 7.5: A syntax diagram of *eventsState*.

Even though an inner condition like the *stateCondition* is a type of condition and in the TA model it represents the condition part of the trigger, it is not generalized by the *conditionNodeType*. The reason for this is that an inner condition is not a separate node but a part of a state node in Node-RED. It consists of the attributes of a state node that describe the characteristics of a condition. Figure 7.6 shows the structure of a *stateCondition*, which is a type of inner condition. The three *halt* non-terminals combined build a normal if clause. The value of the *output* non-terminal determines if the condition is valid. For most state nodes, the attributes contained in the inner

condition are part of the node even if no condition is used. In that case, those attributes are empty. To easily determine if the condition exists, the output attribute is inspected. If it is larger than one, meaning that the node has multiple output ports, then there is a valid condition.

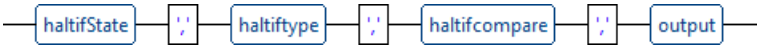

Figure 7.6: A syntax diagram of *stateCondition*.

The nodes in Node-RED that can be used as conditions are generalized by the *conditionNodeType*. Those types represent the condition part of a trigger, like the inner conditions. The *functionCondition* is one of those types. As it is a separate node in Node-RED, in contrast to the inner conditions, the *functionCondition* is a lot more complex. Figure 7.7 depicts the syntax of a *functionCondition*. It has the main attributes that each of the nodes in the grammar has, nevertheless, it contains its own specifics - the *func* non-terminal. This part of the object contains the whole functionality of the function node in Node-RED. The idea behind the function node is to allow a JavaScript code to be used in the rule. The node receives the message from the previous node and can use it and its characteristics(e.g. payload) in any JavaScript function [Opeg]. There is only one condition for the body of the function node, and it is that it must return a message object. The convention for the message object is that it must have a payload attribute. Listing 7.1 shows a very simple JavaScript code that can be added to a function node. The code on line 1 changes the contents of the message's payload. This leads to the next node in the sequence receiving the new payload. The code added to the function can be more complex than the given example. Because the

Listing 7.1 Example of simple JavaScript code that can be added to a function node in Node-RED.

```
1 var msgNew = { payload: 'new payload' };
2 return msgNew;
```

function node can include any JavaScript code, for the purpose of this translation, it is assumed that it can only contain if-, if-else-, and else-constructs that return the message to a specific output port. Nevertheless, adding the syntax of a JavaScript function to a grammar that already deals with JSON's syntax would have immensely increased the complexity of the grammar. Therefore, the code of the function is defined as a string by the grammar. A separate parser is used for the function string. It is discussed in Section 7.2.

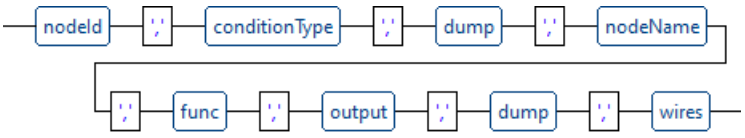

Figure 7.7: A syntax diagram of *functionCondition*.

As can be observed from the Node-RED nodes discussed so far, most of them have similar structures and contain the same basic attributes as *nodeId* and *wires*. Such is the case for the *serviceAction*, as can be seen in Figure 7.8. The *serviceAction* is one of the types that an *actionNodeType* can be. As previously discussed, the *actionNodeType* non-terminals represent the action or actions

in the TA model. The characteristics specific for the *serviceAction* type are the *serviceDomain*, *entityId*, and *service*. The first two provide information about the entity the service is connected to. The *service* non-terminal contains the exact service that will be called.

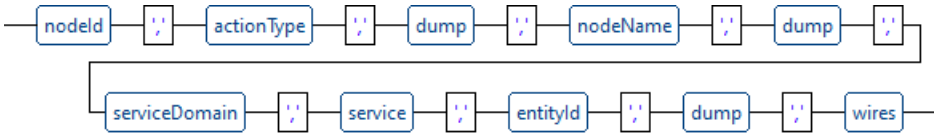

Figure 7.8: A syntax diagram of *serviceAction*.

The last generalized node type, the flow nodes, here represented by the $flowNode$ non-terminal, can be added between the other types. Since they only manage the flow of the rule they do not influence the TA model. Because of this, the $flowNode$ types do not have any specific non-terminals.

7.2 Parsing and Building an Abstract Syntax Tree

With the grammar created, the next step is to implement a parser that recognizes its input using the grammar. ANTLR4 is used to build the parser, as it can automatically generate a parser and a parse tree. When ANTLR4 is used, it also automatically creates classes, called Listener and Visitor, that can access the tree and add functions that are executed when a specific node of the tree is reached. As the parse tree includes a lot of information that is irrelevant for the creation of the textual form of a rule, the functions of the Listener class were not used to directly output the text. Another structure that could be used to print the text was needed.

An AST was chosen as this additional structure. ASTs are usually used as a way to represent the syntactic structure of a source code [BYM+98]. They can show the syntactical structure of and provide lexical information about the code without having to include all of the details of the code, e.g. punctuation [ZWZ+19]. Because of those characteristics, an AST can be used to store a rule without having to store all of the rule's details as a parse tree does. The AST provides the rule's syntax, which can be directly used to build a text. Knowing the meaning behind each node allows one to easily create a sentence around the rule by adding connecting words.

7.2.1 Structure of the Abstract Syntax Tree

The structure of the rule AST is shown on Figure 7.9. It is structured after the TA model, taking into account that a rule can contain multiple TA components. Analogous to the grammar, in the AST, a rule is built out of TA components.The TA node is connected to all partially parallel TA components(if-,else-if-, and else-clauses of a condition). Each TA has a $Filled$ attribute, which shows if some of the parts of the TA have already been added to the tree. It also includes an attribute called $States$, where the ids of the state nodes that are part of the triggers are added. In the TA nodes, the $NextWires$ attribute holds the ids of the Node-RED nodes that come after the state nodes in the original node sequence of the rule.

An important part of the tree are the $IfBlocks$ and the $ElseBlock$. They, together with their attributes, represent the TA components. A TA has at least one $IfBlock$. The $IfBlocks$ are used for if-clauses but also for else-if-clauses. Therefore a single TA can have multiple $IfBlocks$. In contrast, it can have only one $ElseBlock$, which represents the else-clause. The $ConditionId$ attribute stores the id of the condition node of which the $IfBlock$ is part in Node-RED. An $IfBlock$ has at least one $Case$, which incorporates the inner syntax of a comparison in a clause. Each $Case$ has a Bt, which is the boolean operator between this $Case$ and the previous, as a clause can contain multiple comparisons. It also has a $State$, which holds the id of the HA entity, an $Operator$, which is the comparison operator(e.g. ">"), and a $Value$, which is the value used for the comparison. An $IfBlock$ also has at least one $Action$. The $Actions$, of course, are the action part of the TA component. An $Action$ includes the $EntityId$ of the HA entity, which will be activated, and the specific service that will be called($Service$). An $IfBlock$ can be followed by

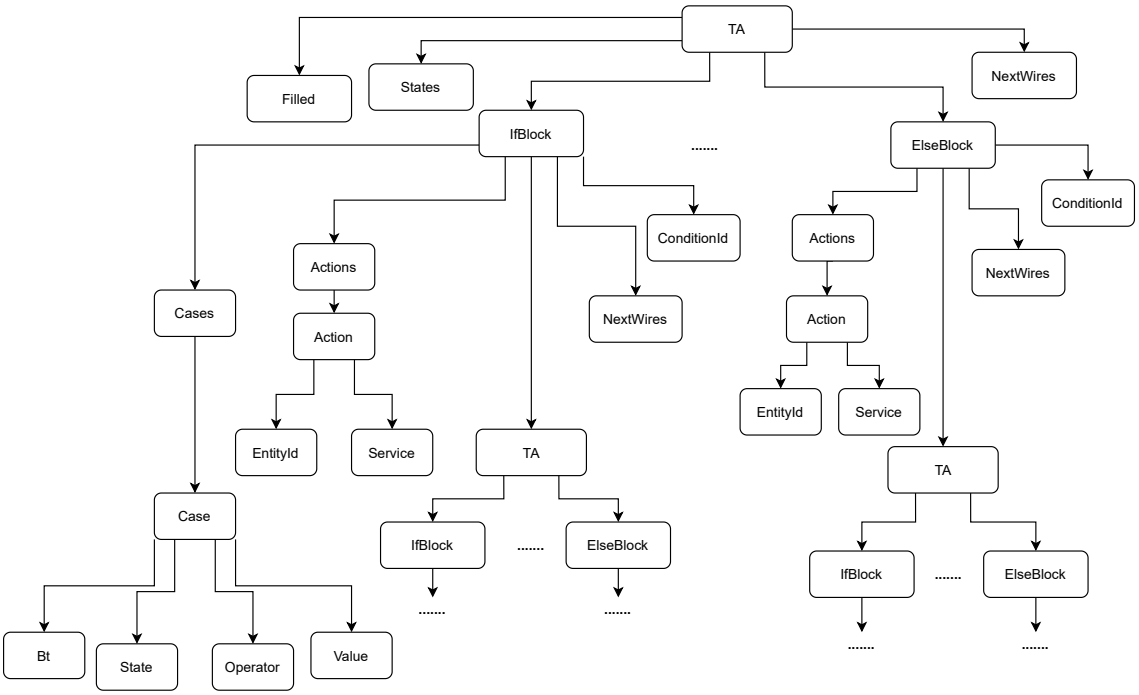

Figure 7.9: A diagram depicting the rule AST.

another TA, which starts the next level of TA components in the rule. The next level has the same construction as the initial one, and its $If-$ and $ElseBlocks$ can be followed by other TA, meaning that there can be a next level of TA components after that. There is no maximal height the three can reach. An $ElseBlock$ is structured similarly to an $IfBlock$, but it has no $Cases$. Again it has a $ConditionId$, and at least one $Action$. As the $IfBlock$, it can be followed by another TA. The $If-$ and $ElseBlocks$, like the TA, have a $NextWire$ attribute. It stores the ids of the Node-RED nodes, which are next in the rule after the nodes whose characteristics have already been added to the AST.

7.2.2 Adding Nodes to the Abstract Syntax Tree

The Listener that is auto-generated by ANTLR4 provides only empty functions for when a node has been entered or exited while the parse tree is being walked. This Listener can be extended to execute specific code when a node of the parse tree is reached. Using this, the parts of the parse tree's nodes that are relevant to the translation can be directly added to the AST. Each of the four types of nodes - state, action, condition, and flow is added in a different way from the others to the tree.

In the parse tree, a state node is the starting point of a whole rule but also of some of the TA components afterwards. This influences the way the node's characteristics are added to the AST. They are added to the TA node, which is the first node that is added when a new level of TA components is added to the AST. The only parts of a parse tree's start node that are added to the AST are its id, its list of wires, and at times its entity id, which references the HA object from which it receives its data in Node-RED. The node id is needed to keep track of the state nodes of the parse tree that have been added. The wires are important for determining where the characteristics of the next nodes in the rule will be added. Some of the condition nodes in Node-RED and then in the parse tree contain all of the information needed to define triggers in the textual representation of the node. Others do not and need the entity id of the state node. Where and why the entity id attribute of a state node is added to the AST is discussed in the following paragraphs.

When the state node's attributes are added to the AST, the first thing is to determine where exactly they will be added to the tree. Algorithm 7.1 shows an algorithm that finds the right place for a state node's characteristics. The check starts at the root rule of the tree and goes downward. First, it is checked if the current TA is filled(line 4). For a TA to be filled, it means that the parts of it representing its trigger, meaning all of the state and condition nodes' attributes, have been added already. If that is not the case, the state's attributes can be added to this TA. Then the current TA is returned by the algorithm(line 5). If the TA is filled, it is not possible to add the state's characteristics to it, which means the search should go a level down. To determine which edge to follow, the TA node's blocks are inspected. Each $IfBlock$ is checked to see if the state node's id is in its $NextWires$(lines 7-14). If it is, the $IfBlock$ node's child TA becomes the current TA(line 12). If the id is not in any of the $IfBlocks$, then the $ElseBlock$ is checked(lines 15-22). Identically to the $IfBlock$, if the id is in the $ElseBlock$ node's $NextWires$, the $ElseBlock$ node's child TA is set as the current TA(line 20). This is repeated until a current TA if found that is not filled.

After the place for the state's characteristics has been determined, the state node's id is added to the $States$ of the chosen TA. If the state node had an inner condition, the state's wires are not added to the TA. If that is not the case, the wires are added to the TA's $NextWires$ and are propagated throughout the $If-$ and $ElseBlock$ nodes that are part of the path from the TA to the root of the tree.

The purpose of the propagation is to shorten the search for the path to where the next node's characteristics can be added. The idea is to have the $NextWires$ of the leaves of the tree in all of $If-$ and $ElseBlock$ nodes that are the parent, parent of the parent of the leaves, and so on until the root is reached. When an element needs to be added to the leaf, it is easy to determine where to add it. When the wires of a node are propagated, the id of the node is removed from the $NextWires$ of the blocks and TA nodes so that the $NextWires$ list does not become cluttered with needless ids.

Algorithm 7.1 Function that finds the place for a state in the AST.

```
1:  procedure FIND_STATE_PLACE(id)                          // the state node's id
2:      initialize last_ta = root                           // the AST's root
3:      while True do
4:          if last_ta is not filled then
5:              return last_ta
6:          else
7:              for IfBlock in last_ta.IfBlocks do
8:                  if id in IfBlock.NextWires then
9:                      if IfBlock does not have a child TA then
10:                         create child_ta
11:                     end if
12:                     last_ta = child_ta
13:                 end if
14:             end for
15:             if last_ta.ElseBlock exists then
16:                 if id in last_ta.ElseBlock.NextWires then
17:                     if last_ta.ElseBlock does not have a child TA then
18:                         create child_ta
19:                     end if
20:                     last_ta = child_ta
21:                 end if
22:             end if
23:         end if
24:     end while
25: end procedure
```

As with the state, when parts of a condition node are added, first their place needs to be determined. The parts of a condition node that are added to the AST are its if-, else-if-, and else-clauses, its wires, and the node id. The if and else-clauses are added to the tree separately because in the tree they are represented by the $If-$ and $ElseBlocks$. The else-if-clauses are added identically to the if-clauses and are represented by $IfBlock$ nodes as well. The block nodes represent the trigger part of the TA component. The way their place in the tree is found is identical to how the state attributes' place is determined. When an if statement is added to the tree, a new $IfBlock$ is created. All of the comparisons of the if statement are added as $Cases$ to the block. The wires of the output port of the if-clause are added to the $NextWires$ of the $IfBlock$ and are then propagated. When the if-clause is the last clause of the condition, the $Filled$ flag of the TA to which the $IfBlock$ has been added is set to true. Additionally, the state nodes' ids are removed from the $NextWires$ of all of the nodes leading from the current TA to the root node. The method of removal is similar to the propagation. The ids are removed from all blocks on the path to the root. When an else-clause is added, a new $ElseBlock$ is created. The wires of the output port of the else-clause are added to the $NextWires$ of the $ElseBlock$ and are then propagated. Since the else-clause is always the last one in a condition, the $Filled$ flag is set to true. Furthermore, as with the $IfBlock$, the ids of the states in the TA are removed from the $NextWires$ of the current TA node's parent and all of the nodes after it leading a path to the root node.

The method for finding an action's place in the tree is a bit different from that of the state and condition nodes. The main difference comes from the fact there is an order to how the elements are added to the AST. Each element is appended when its corresponding node in the parse tree is traversed. Because of the structure of the parse tree and the fact that with ANTLR it is traversed in a depth-first manner, the order in which an action is added to the AST is always after a condition's characteristics have already been added. This means that when an action is added, the TA node is already filled and all of the $If-$ and $ElseBlocks$ have already been created. An action node from the parse tree corresponds to the $Action$ element in the AST. The difference between them is that in the AST the $Action$ element only contains the entity id of the HA entity that is controlled by the action and the service that is executed during the action.

As with the previous functions that add elements to the AST, the first thing that is done is to search for a place for the action. Algorithm 7.2 shows an algorithm that searches for it. Starting from the root TA, it first examines the $IfBlocks$, checking if the id of the action node is in one of the $IfBlock$ nodes' $NextWires$(lines 2-10). If that is the case and the $IfBlock$ does not have a child

Algorithm 7.2 Function that finds a place for an action in the AST.

```
1:  procedure FIND_ACTION_PLACE(id, ta)// the state node's id and the current TA that is checked
2:      for IfBlock in ta.IfBlocks do
3:          if id in IfBlock.NextWires then
4:              if IfBlock does not have a child TA then
5:                  return IfBlock
6:              else
7:                  call this procedure with IfBlock's child TA
8:              end if
9:          end if
10:     end for
11:     if ta.ElseBlock exists then
12:         if id in ta.ElseBlock.NextWires then
13:             if ta.ElseBlock does not have a child TA then
14:                 return ElseBlock
15:             else
16:                 call this procedure with ElseBlock's child TA
17:             end if
18:         end if
19:     end if
20: end procedure
```

TA, this $IfBlock$ is the right place for the action(line 5). If, however, the checked $IfBlock$ has an existing child TA, then the same procedure is done for the $IfBlock$ node's child TA(line 7). This is repeated until a block is reached that does not have a child TA and contains the node's id. Provided that the id was not found in any of the $IfBlocks$, the $ElseBlock$ of the root is examined(lines 11-19). If its $NextWires$ contain the id, similar to the $IfBlock$ search, it is checked if the $ElseBlock$ has a child TA(lines 13-17). Assuming that is not the case, the $ElseBlock$ is returned(line 14). Supposing a child TA exists, the function is executed with the child TA(line

16). Again this is repeated until the right block is found. When the action's place is found, a new *Action* is created and added to the block. Its wires are also added to the block and then propagated, removing the id of the action node whilst this is done.

As flow nodes are not fully part of the rule model, they can potentially be added at different places in the rule. This means that a flow node can be added to a TA like a state's attributes are but can also be added to an $If-$ or an $ElseBlock$ like a condition's clauses or an action can. Because the flow nodes do not add any meaning to the TA components, only their wires are added to the AST. This is done to keep the right order of the TA components in the initial rule. Algorithm 7.3 shows how the AST is searched to find the right place for a flow node's wires. The function starts at the root TA. First, it is checked if the TA has any IfBlocks(lines 2-4). If it does not, then the flow node's wires will be added to the TA(line 3). In case there are $IfBlocks$, then they are examined until an $IfBlock$ is found that contains the node's id in its $NextWires$(lines 5-13). Similar to the algorithm for an action, if the $IfBlock$ has a child TA, then this function is used with the child TA(line 10). This is done until a TA or a block is returned. In the case where the $IfBlock$ does not have a child TA, then this block is returned(line 8). When the id is not in the $IfBlock$, the same check is done for the $ElseBlock$(lines 14-22). When the right place for the flow node's wires

Algorithm 7.3 Function that finds a place for a flow node in the AST.

```
1:  procedure FIND_FLOW_PLACE(id, ta) // the state node's id and the current TA that is checked
2:      if ta does not have IfBlocks and id not in ta.NextWires then
3:          return ta
4:      else
5:          for IfBlock in ta.IfBlocks do
6:              if id in IfBlock.NextWires then
7:                  if IfBlock does not have a child TA then
8:                      return IfBlock
9:                  else
10:                     call this procedure with IfBlock's child TA
11:                 end if
12:             end if
13:         end for
14:         if ta.ElseBlock exists then
15:             if id in ta.ElseBlock.NextWires then
16:                 if ta.ElseBlock does not have a child TA then
17:                     return ta.ElseBlock
18:                 else
19:                     call this procedure with ta.ElseBlock's child TA
20:                 end if
21:             end if
22:         end if
23:     end if
24: end procedure
```

is found, the node's id is removed from the $NextWires$ of the node no matter if it is a block or a TA. Afterwards, the node's wires are added to the TA or block and are propagated whilst the node id is removed from the tree.

When the parse tree is traversed, each of the Node-RED nodes is added in a different way in accordance with its structure. When the tree is walked, and for example, one of the state nodes is reached, there are two possibilities for adding the characteristics of the node to the AST. If the state node has one output port, meaning that the node does not have an inner condition, all of the state node's attributes that were previously mentioned are added to the tree. When the node has multiple output ports, this means that the node has an inner condition. In that case, when the function that adds a state's characteristics to the tree is used, it is not given any of the node's wires. This is done because they are also the wires of the inner condition and will be added when the condition clauses are added.

When a condition is added, the traversing order of the tree is used. Before the condition is appended, its cases(the comparisons) are prepared so when the condition node is exited all of the prepared data can be used to call the functions that add the if-, else-if-, and else-clauses. A specificity of inner conditions is that when their clauses are added to the tree, they use the entity ids of their state nodes. This is done because the inner conditions do not contain the entity id themselves. This is possible as both nodes in the parse tree are defined in such a way that the state node is the inner condition node's parent. This connection allows both nodes to be able to access each other's attributes.

As mentioned earlier in Section 7.1, when the condition is a function node, the process of parsing it and adding it to the AST is more complex. A separate parser is generated for the JavaScript code contained in the function. The parser is auto-generated from a grammar created to support the JavaScript syntax and the assumed syntax of a function. The grammar of the function can be found in Appendix A.2. The grammar expects that there are only if-, else-if-, and else-clauses in the function. As with the original parser, a parse tree is built when the input is parsed. The auto-generated Listener is extended so it can deal with adding the clauses to the AST. The conditions of each of the clauses are gathered and are handed over to the functions that append the clauses to the tree. When the whole function parse tree has been traversed, the updated AST is returned to the initial rule Lister which then continues to add the other nodes of the rule to the AST.

Adding action and flow nodes' characteristics is quite trivial. When one of those two node types is reached while the parse tree is being traversed, they are added by simply calling their respective functions. The specific characteristics that are needed for the AST are given directly to the function. There are no particularities as with the other node types. The functions are provided with all of the nodes' wires regardless of the number of output ports.

7.3 From Abstract Syntax Tree to Text

When the AST has been filled, a textual representation of the Node-RED rule needs to be created from it. The process of assembling the rule text is comprised of multiple interconnected steps. The steps are repeated for each TA and its blocks($IfBlocks$ and $ElseBlock$). Step 1 is determining which blocks in a TA have been triggered. Step 1.1 is retrieving the condition for each block and checking if the current values fulfill it. If the block that has been triggered is an $ElseBlock$, there is an additional step before the second step (step 1.2). In this preliminary step, the conditions of the $ElseBlock$ are prepared. Step 2 deals with the triggered blocks, processing them one by one. The first part of step 2(step 2.1) is assembling the text for each of the $Cases$ in the block. Step 2.2 is putting together the text for each of the $Actions$ in the block. The last part of step 2(step 2.3) is assembling a sentence for the block from the parts assembled in steps 2.1 and 2.2.

The model defined for the rules, states that a rule can have multiple nested TA components. Furthermore, in one rule there can be multiple partially parallel TA components since else-if- and else-clauses are allowed. When all of the possible paths that can be triggered are added to the text, it can become too complex and incomprehensible when a more complex rule is translated. Therefore, it is better for the readability of the text to add only the parts of the rule that have been triggered when the rule was executed. Triggered means that their conditions have been fulfilled, and the actions connected to the trigger were executed.

That is the main purpose of the first step, to determine which parts of the rule have been triggered. Algorithm 7.4 shows how this is accomplished. The function traverses a rule AST. For a given TA node, it checks which $If-$ or $ElseBlock$ was triggered. This means determining which of the if-, else-if-, or else-clauses are fulfilled. The exact method for doing this is explained in the following paragraph. The traversing starts with the root TA and works its way downward. The

Algorithm 7.4 Function that starts the process of assembling the text of a rule.

```
1:  procedure PRINT(nextNode)                                   // a TA node
2:      if nextNode exists then
3:          initialize printed_if = false// flag that shows in an IfBlock has already been added
    to the text
4:          if nextNode has IfBlocks then
5:              initialize num = 1 // shows how many of the IfBlocks have been added to the text
6:              for IfBlock in nextNode.IfBlocks do
7:                  if condition of the IfBlock is satisfied then
8:                      pass IfBlock, num, nextNode, and IfBlock.Cases to print_helper
    procedure
9:                      printed_if = true
10:                     num++
11:                 end if
12:             end for
13:         end if
14:         if not printed_if and nextNode.elseBlock exists then
15:             get all IfBlock conditions and invert them
16:             pass nextNode.ElseBlock, nextNode, and the inverted conditions to
    print_helper procedure
17:         end if
18:     end if
19: end procedure
```

$IfBlocks$ are examined first(lines 4-13). When it is found that one of them has been triggered(line 7), it is passed to the next step of the printing process (step 2)(line 8). The number of $IfBlocks$ that have been triggered is tracked since it is relevant to how the text will be constructed(line 10). If even one $IfBlock$ has been triggered, the $ElseBlock$ of the TA is not considered at all. If none of the $IfBlocks$ were triggered and an $ElseBlock$ exists, then it must be the part of the TA that was executed. When that is the case, all of the $Cases$ of the $IfBlocks$ are gathered and inverted so a condition for the $ElseBlock$ can be established(line 15). The process is explained in a later paragraph. Subsequently, the $ElseBlock$ and its new condition are then handed over to the next step(line 16).

This paragraph explains in more detail how it is determined which $IfBlock$ was triggered. The check is conducted by using the last values of the sensors that triggered the rule. These values were sent in the HTTP request when the rule was triggered. For each of a block's cases, a real condition is constructed using the real values and the $Case's\ Operator$ and $Value$. In accordance with the boolean operators between the cases, the whole condition of the block is checked.

In this paragraph, it is discussed how the $Cases$ for the condition of the $ElseBlock$ are gathered and prepared. Those processes are performed separately. The $Cases$ of the $ElseBlock$ are the inverted versions of the $Cases$ of all of the $IfBlocks$ in the TA. When the $Cases$ are gathered a null object is added between the $Cases$ of different $IfBlocks$. In the procedure that inverses the conditions, for each of the $Cases$, a new $Case$ is created. The new $Case$ has the same $State$ and $Value$ as the initial one, but its $Operator$ is the opposite of the original one. After this, the boolean operators(Bt) between the $Cases$ are also reversed. When this process is complete, the $Cases$ are added to a list. As in the original list, there is a null element between the $Cases$ of different $IfBlocks$.

After a block($If-$ or $ElseBlock$) that has been triggered is found, it is given to the function shown in Algorithm 7.5, which implements the second step. First, in step 2.1(lines 2-5), the text for each of the block's $Cases$ is built. The mechanics of this are explained in a later paragraph. When the text of a $Case$ has been formed, it is added to a list that only contains the $Cases$ of the current block. Afterwards, the $Actions$ of the block are examined and their text is assembled. This is the process contained in step 2.2(lines 6-8). As with the previous step, its details are explained later. Identically to the $Cases$, when an $Action's$ text is composed, it is added to a list that has the texts of all of the $Actions$ of the current block. When all $Actions'$ texts have been gathered, the $Cases'$ and $Actions'$ texts are combined to form a sentence. This is step 2.3(line 10). The specifics behind it are discussed in the last paragraph. When the whole sentence has been constructed, it is appended

Algorithm 7.5 Function for building a text from a block.

```
1:  procedure PRINT_HELPER(node, block, cases, num)   // node = the current TA, block = the
    current block, cases = the Cases of the block, num = flag that shows how many of the blocks
    have been added
2:      for case in cases do
3:          get text for case
4:          add text to case_text // global variable that temporally holds the text of all Cases of a
    block
5:      end for
6:      for action in block.actions do
7:          get text for action
8:          add text to action_text // global variable that temporally holds the text of all Actions
    of a block
9:      end for
10:     combine case_text and action_text and add them to the text of the rule
11:     call print(nextNode) with the child TA of block
12: end procedure
```

to the text of the rule. Subsequently, the print(node) procedure is called for the child TA of the current block. This process is repeated for all of the TA nodes in the AST.

This paragraph explains the specifics of step 2.1. HA provides a REST API that supplies additional information for a given entity that is connected to it. As the $Cases$ contain the entity ids of the sensors that are used for the condition, additional information is requested for each $Case$. The provided information contains a friendly name for the entity, which is then used in the text. The friendly name is the name the user has chosen for the entity in HA. When it is used in the text, the user can understand more easily which entity is part of the rule. Furthermore, with the additional information, it can be determined what type of data is outputted by the sensor. If the output is not boolean, a unit of measure is also supplied by HA. When the output is boolean HA provides the class of the entity. With this information, the words that go with the values are determined. Moreover, the $Operator$ of the $Case$ is swapped for its textual representation. Using all of the gathered segments, a text is build, adding some words to fill gaps between them.

In this paragraph, it is explained how the text of each $Action$ is build(step 2.2). As with the $Cases$, the additional information about the actuators is retrieved from HA's REST API. Because HA does not provide information about the type of actuator that is used, it is assumed that hints about its type can be found in the friendly name or the entity id of the actuator. If that is not the case, then the basic phrases "turned off" and "turned on" are used when saying what the $Action$ has done.

The way the $Cases'$ and $Actions$' texts are connected in a sentence(step 2.3) is discussed in this paragraph. The role of this step is to add a meaningful beginning to the sentence. Additionally, it fills the gaps between the text segments. The beginning of the sentence is dependant on the num variable(line 10 in Algorithm 7.4). It shows how many $IfBlocks$ of the same TA have been triggered. That number is important because if multiple blocks have been triggered, it makes sense to start their sentences differently. Afterwards, the $Cases$ are added, keeping the order of their list. After adding a phrase behind the $Cases$, the $Actions$ are also attached to the text.

8 Collection of User Feedback

After an automation rule has been translated into a human-readable form, the next steps are to research how this now readable rule can be used to dynamically change rules using users' feedback. For that purpose, the question of how the feedback is collected from the users is crucial for this research. The method for collecting users' feedback to a triggered rule is discussed in this chapter. Section 8.1 explains how the human-readable rule is sent from the translation component to the UI component. The significance and usage of the rules' location are explored in Section 8.2. In Section 8.3 the UI that is used for gathering the users' feedback is examined. The manner in which the human-readable rule is presented to the users is shown, and the method for collecting the feedback is described in that section. Finally, Section 8.4 talks about how the feedback is used to improve the rules. First the method for processing the feedback is presented. Additionally, an approach for recognizing specific parts of the rule that are to be changed is explained, and then the manner in which these parts are changed is presented.

8.1 Sending the Feedback to the User Interface

When an automation rule has already been translated, the next step is to make the rule available to the users of the system. This way they can provide their feedback and with it potentially change some aspects of the rule. As mentioned in the previous chapters, the translation process is initiated when the server of the translation component receives an HTTP request from Node-RED. After the translation has been completed, the human-readable rule is returned to the function that handled the HTTP request in which the information about the triggered rule was received. It is not sufficient to just send the rule text to the UI. There needs to be additional information that gives the users more context about the rule that has been triggered. Furthermore, some details about the conditions in the rule are also necessary. They give the users the option to change specific aspects of the rule and not just state if they like the rule or not. For that purpose, when the textual representation of a triggered automation rule is returned, a specific JSON structure is composed. The structure can be seen on Listing 8.1. The first attribute of the JSON object - "id"(line 2) is a the id of the rule. The condition id of the first $IfBlock$ of the AST is used for this. The next attribute(line 3) is the text of the rule. After that, there are attributes called "current_state" and "case_groups". The "current_state"(lines 4-14) contains the object that was received from Node-RED containing the last values of the sensors that were part of the triggered rule. The "case_groups"(lines 15-37) attribute includes a list of JSON elements. Each element represents one if-, else-if-, or else-clause. Each condition comes from an $If-$ or an $ElseBlock$ in the AST that has been triggered. It contains the id of the Node-RED node of which the clause is part of and all of the clause's comparisons. The comparisons are themselves a list of JSON objects. Each comparison includes the structure of a $Case$ in the AST. Additionally, it includes the friendly name of the sensor and a boolean flag that indicates if the sensor is a boolean sensor. The last attribute in the JSON object(line 38) will be explained in the next section.

Listing 8.1 Example structure of the JSON object sent to the UI.

```
{
   "id":"154a909c.41f527",
   "text":"When lightsensor detects a value that is lower than 55 % and motionsensor1 detects motion
, then Lamp will be turned on.",
   "current_state":{
      "entity_id":"None",
      "payload":{
         "binary_sensor.motionsensor1":"on",
         "sensor.lightsensor":"4"
      },
      "ids":[
         "8016aed6.8641f8",
         "962511f3.3bf3f"
      ]
   },
   "case_groups":[
      {
         "node_id":"154a909c.41f527",
         "condition":[
            {
               "state_id":"sensor.lightsensor",
               "operator":"<",
               "value":"55",
               "nice_state":"lightsensor",
               "nice_value":"55",
               "is_bool":false
            },
            {
               "state_id":"binary_sensor.motionsensor1",
               "operator":"==",
               "value":"on",
               "nice_state":"motionsensor1",
               "nice_value":"motion",
               "is_bool":true
            }
         ]
      }
   ],
   "areas":['Kitchen']
}
```

Following its assemblage, the JSON object is send to the UI component. Because the scope of this thesis's research includes systems with multiple users, the human-readable rule must be sent to all of the users of the system. For this purpose, it is not sufficient to send the JSON object to a single UI. Therefore the publish-subscribe pattern was chosen for the task. As the setup of the system already included an MQTT broker, which incorporates the chosen pattern, it was the technology of choice for the connection between these two components. Using Eclipse Paho's MQTT library for Python, an MQTT client was added. Every time a triggered rule is translated, the JSON object containing the text is published to the MQTT broker. From there, each device that has the UI component can subscribe to the broker and receive all of the rules.

8.2 Rule Location

When a rule is triggered, apart from its translation and characteristics also its location in the home is of importance. The reason for this is that the users whose preferences are most important are the ones that are in the same space as the triggered rule. Therefore, it is important to send the triggered rules to users that are currently in the same area as the sensors and actuators that are part of the rule. For this, the locations of the devices that are part of the rule and the users need to be determined.

The location of the entities and actuators is asserted using a HA concept called Areas. In HA, entities cannot have Areas, but their devices can. The entities used in this thesis were mocked, so their devices also had to be mocked. This was done by adding a device to the sensors' configurations in HA's configuration file. Afterwards, the devices had to be discovered by the system. This was accomplished by sending MQTT messages to a specific topic with the characteristics of the device. When a device is added to HA, its Area can be set.

The discovery of a rule's Area or Areas is completed during the preparation of the JSON rule object. HA's REST API, which was used to retrieve the rules and characteristics of the entities and actuators, does not provide any information on devices or Areas. HA's WebSocket API however, allows to retrieve both. It can supply lists with all areas, devices, and entities added to HA. Using the ids of the entities that are part of the triggered rule, the entity objects were retrieved from the list. The entity objects contain information about the devices they are part of. Using that information, it was possible to get the devices for those entities. As the entity objects, the device objects in the list carry the id of the Area they are in. With that id, the Area of the device is extracted from the list. This is done for all entities in the triggered rule. The Areas are then added to the JSON object that will be send to the UI component.

8.3 Feedback User Interface

The UI is the place where the users can see the textual representation of an automation rule. There, they can judge the effectiveness of a rule and provide feedback for the improvement of the rule. As previously discussed in Section 5.1, the UI component is implemented as an Android application.

The first thing the application does is ask the users for their location. The purpose of this is to determine which rules should be shown to the users. When the application is opened, it shows them a list of the available Areas, and the users have to select one. The Area list is retrieved from the back-end by sending an HTTP request. The request is sent to the same server via which a triggered rule is delivered from Node-RED to the translation component. When the request is received, the Areas are acquired by using HA's WebSocket API. All of the possible Areas in HA are then sent back to the UI component in the response of the HTTP request. The view where the Areas are shown can be seen in the first figure of Figure 8.1.

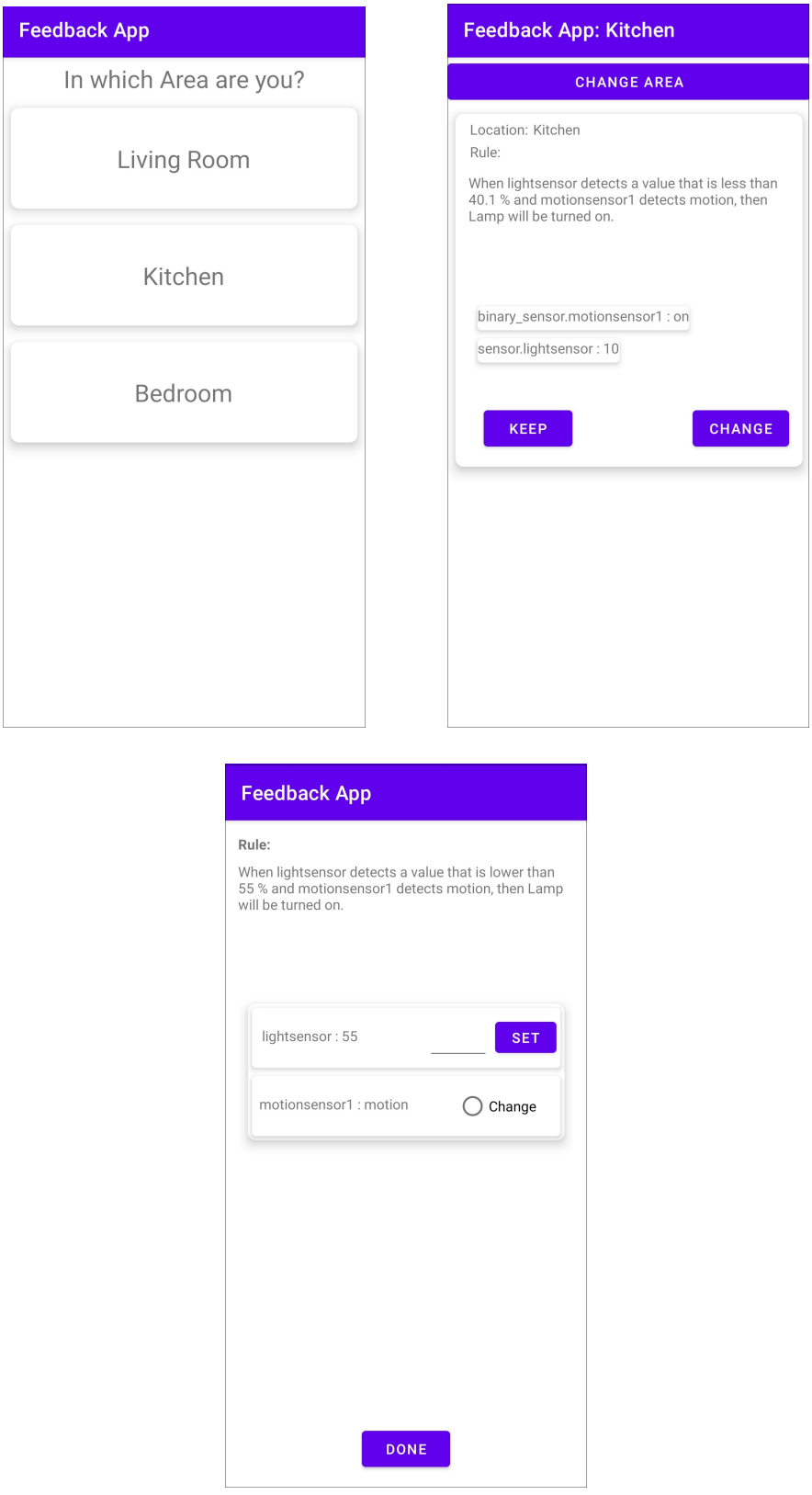

Figure 8.1: The visual representation of the UI component.

After an Area has been selected, the rules that have been triggered for this Area can be shown. Similarly to the translation component, here, an MQTT client implementation is used to subscribe to the MQTT broker. Doing that, every time a rule is triggered and translated, it will be received by the application. When the JSON rule object is received, the first step is to map it to a Java object so it can be used more easily. The models used for the data that is received are shown in Figure 8.2.

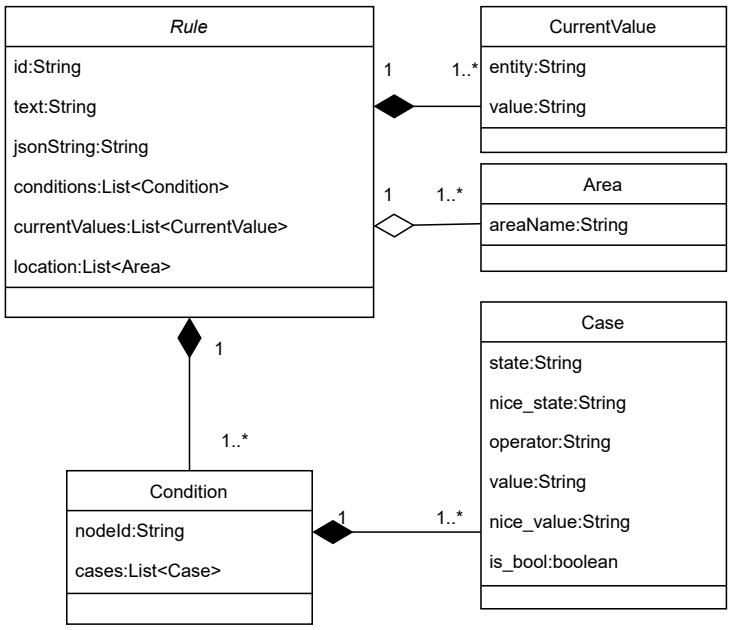

Figure 8.2: Class diagram of the model for the rule in the UI component.

Each Rule has an id, a text, and contains the whole JSON received from the translation component. It also has a list of conditions, which represent the if-, else-if-, and else-clauses. A Condition is built the same way it was in the received JSON object. It has an id and a list of Cases. The Cases also have the same structure as before. Additionally, the Rule model contains a list filled with the current values of the sensors that are part of the triggered rule (CurrentValues). Furthermore, the Area or Areas that the rule's entities are in, are also added to the model in the list attribute called location.

When the received rule has been mapped to its Java model, it is checked if its Areas contain the Area that the user has chosen. Then it is added to a list containing all other rules to which the user can give his feedback. How a rule is displayed can be seen in the right picture in Figure 8.1. Additionally to the text of the rule, the values of the sensors that are part of the rule are also displayed. The shown values are the ones that triggered the rule. They are displayed so that the users can have additional information to help them form their opinion.

For each rule, the users have two possibilities for giving feedback. The first one is to give positive feedback, leaving the rule the way it is. The other is to give negative feedback by changing some of the rules aspects. When the "keep" button is clicked, the user gives positive feedback, and the rule is removed from the list. Additionally, its values are sent to the component that updates the rules. If the "change" button is clicked, the users are led to another view where they can choose what to change. The properties that the users can modify are the values used for the comparisons in the if-, else-if-, and else-clauses. The bottom picture in Figure 8.1 depicts the part of the application that provides this feature. In comparisons that do not use boolean values, the users can add their desired

value themselves. When the value is boolean, they can only state that they want a change. When the users are done with their changes, they have to click the "done" button. Subsequently, the rule is removed from the list with rules ready for feedback, and the changes the users want are sent to the component that is responsible for updating the rules. Afterwards, the users are returned to the list of triggered rules for the Area they have chosen. There, they can give their feedback for other rules.

When a rule is reviewed, a new JSON object is created. The object is needed since the feedback needs to be sent back to the back-end, where it can be applied to Node-RED. Listing 8.2 presents a simpler version of the object originally received from the translation component. The attributes contained in the JSON object are almost the same as with the previous JSON, except that there is no "current_state" attribute and there is an additional attribute called "user_id"(line 3) where the id of the user's device is stored. This is used to determine which feedback comes from which user. The "case_groups" attribute is renamed to "conditions" and the "condition" is now called "cases". The attributes of a case in "cases" have the same structure as before except, here, each case has only a "state_id", "operator", and "value"(lines 8-10 and 13-15). If the user has made any changes, they are stored in the "value" attributes. The JSON is send directly to the updating component using an HTTP request.

Listing 8.2 Example structure of the JSON object sent to the Component for Updating Rules.

```
1  {
2     "id":"154a909c.41f527",
3     "user_id":"0d624eb7279fxxxx",
4     "conditions":[
5        {
6           "cases":[
7              {
8                 "state_id":"sensor.lightsensor",
9                 "operator":"lower",
10                "value":"55"
11             },
12             {
13                "state_id":"binary_sensor.motionsensor1",
14                "operator":"equal",
15                "value":"on"
16             }
17          ],
18          "node_id":"154a909c.41f527"
19       }
20    ]
21 }
```

8.4 Correcting a Rule using the Feedback

The role of the component for updating rules is to apply the changes received from the UI component to the corresponding rule defined in Node-RED. Because the system can have multiple users, a user's feedback cannot be directly used to change a rule. A compromise must be found for the wishes of all users. Such a compromise is calculating the mean of the feedback of each of the characteristics of a rule and using those values when updating the rule.

The HTTP request from the UI component is received by the same Python server that received the initial request by Node-RED. The JSON contained in the request is added to a database that contains the feedbacks. As previously mentioned, the database used for this is InfluxDB. The manner in which the feedback is added to the database is shown on Listing 8.3. The id of the rule and the

Listing 8.3 An example of the structure of the entries in the database.

```
1          {
2              "measurement": "changes",
3              "tags": {
4                  "id": "cb1e4024.6e637",
5                  "user_id": "0d6xxxxxxxxxx"
6              },
7              "time": "2009-11-10T23:00:00Z",
8              "fields": {
9                  "conditions": "[{'cases': [{'state_id': 'sensor.tempsensor1', 'operator': 'gte', '
   value': '22'}], 'node_id': 'cb1e4024.6e637'}]"
10             }
11         }
```

"user_id" are added as tags(lines 4-5), so they can be later used in queries. Since the "conditions" in the JSON object are an array that contains arrays and InfluxDB does not provide an appropriate way to store them, they are added as a string. The "time" attribute is the date and time of adding the feedback to the database.

When a user's feedback for some rule is added, it is checked if the database contains feedback for the same rule from the same user. If that is the case, the previous feedback is removed from the database. This is done in order to have only the most recent feedback when calculating the mean. After the feedback has been added, the feedbacks of all users for the specific rule are retrieved. Only the feedbacks for a specified time window are taken since older feedbacks will make the average value not viable. From those feedbacks for each case in each condition, the average value is calculated. If the value is numeric, the mean is used. When it is not numeric, the mode of the value set is determined and used. If there are values that are used equal times, then one of those is randomly selected for the mode.

When the mean or mode for each case value is calculated, it is added to the initial JSON rule object that was received from the UI. Afterwards, it is passed to the component that will update the rules. Using Node-RED's API all of the nodes are retrieved. The nodes are inspected, looking for nodes that have the same id as the "node_ids" in the "conditions" of the JSON object. When such a node is found, the manner in which it is updated depends on its type. If the node is some of the state node types, it means that the condition was an inner condition. Then the change is trivial to make since only the value of one of the node's attributes needs to be replaced. Thereafter, the node can directly be returned to Node-RED.

When the node is of type "function", the process is more complex. Similar to the initial parsing, the complexity comes from the fact that the condition in a function node is not contained in its attributes as with the inner conditions, but is given in the form of a JavaScript code. To change certain characteristics of the code, one should know exactly where they are. Additionally, one should be able to retrieve specific parts of the code so they can be replaced. This is achieved by reusing the function parser that is used in the translation component. The function is parsed, and

automatically a parse tree is built. Using the parse tree, a map that contains the values for each case is assembled. The map uses a combination of the "state_id" and the "operator" as a key. This way, when a case value needs to be changed, the old value for it can easily be retrieved. The idea behind this is to build identical strings, one containing the old value and one containing the new. When function text is updated, the old string is swapped with the new one. This is done for all of the cases contained in the condition.

After all values for the changed nodes have been added, the node list retrieved from Node-RED contains the new values. As the list is made of all of Node-RED's nodes it can be directly returned to Node-RED. When it is returned the changes are applied in the changed rule.

9 Evaluation

In this chapter, the research of this thesis is evaluated, and the limitations of the presented concepts are discussed. First, in Section 9.1, the translation of rules into human-readable text is evaluated. Then in Section 9.2 the collection of rule feedback is assessed. Furthermore, in Section 9.3 the research questions of this thesis are discussed and evaluated. Finally, in Section 9.4, the limitations are presented.

9.1 Evaluation of the Translation

To validate the translation of a rule into human-readable text, the three use cases defined in Chapter 4 were used. The use cases were defined in such a way that they represent different levels of complexity. As mentioned in Section 5.2, Ur et al. [UMPL14] performed a study that shows what types of automation rules regular users want. As previously discussed, the biggest part of the participants of the study used only one TA component for their rules. The first use case was used to determine if such rules can be successfully translated by the translation component. The second and third use cases were used to ascertain if the translation could handle rules that have multiple triggers or actions and multiple TA components. All three use cases were implemented in Node-RED.

Figure 9.1 shows how the first use case was modelled using Node-RED. It was defined in the following way:

- If the temperature sensor's value is above 25 °C, then the fan is turned on. If it is below 25 °C then the fan is turned off.

Because there was only one state node("temp_sensor_1"), instead of using a condition node, the inner condition of the state node was used. Therefore, the two action nodes("fan_turn_on" and "fan_turn_off") were directly connected to the state node that gets data from the temperature sensor. Using the mock sensors implemented for this thesis the rule was triggered and passed to

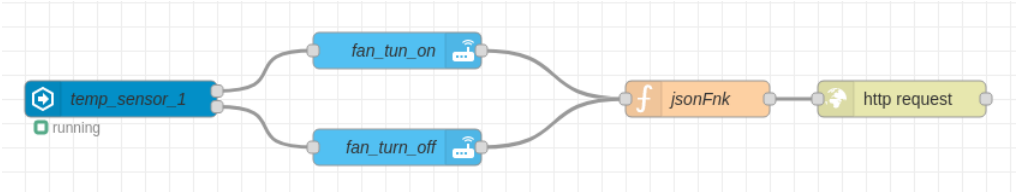

Figure 9.1: The first use case implemented in Node-RED.

the translation component. There, it was fully retrieved from Node-RED and put in the right order that can be recognized by the parser. The rule in its JSON form can be found in Appendix A.3.1. In that form, it was given to the parser, and a parse tree is created from it. The parse tree was

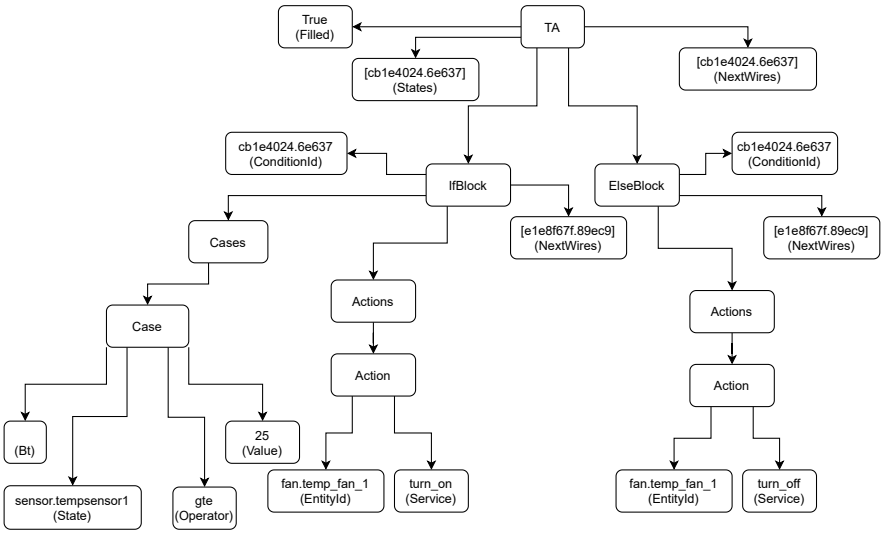

Figure 9.2: The first use case's AST.

then used to build an AST. The AST for this rule is shown on Figure 9.2. As shown in the figure, the AST had only one TA, where the two TA components, the $IfBlock$ and the $ElseBlock$, are contained. The $NextWires$ of both of those are the same because the next node in the Node-RED implementation of the rule is the "jsonFnk" node. The Bt of the $Case$ is empty because there is only one comparison. Bt is the boolean operator that links comparisons together. Even if there were multiple $Cases$, the Bt of the first $Case$ would always be empty. After the AST is used to form the text of the rule, the following output is supplied:

- When tempsensor1 detects a value that is higher than or equal to 25 °C, then Temp Fan 1 will be turned on.
- When tempsensor1 detects a value that is less than 25 °C, then Temp Fan 1 will be turned off.

The resulting text is very similar to how the use case was initially explained. This use case can be used as an example for all rules that contain a TA with only one trigger and one action. This means that for all similar simple home automation rules, the presented solution should be able to return an adequate textual representation.

The implementation of the second use case in Node-RED is shown on Figure 9.3. For this use case, the following conditions were defined:

- If there is motion in the room and the light is less than 40 %, then the lamp should be turned on.

Because the condition for this rule uses the values of more than two sensors, most of the state node types' inner conditions cannot be used. Most state nodes receive data from only one sensor, which means that in their inner conditions only that sensor can be used. Therefore, for this use case, a separate condition node is used. Before the sensor values can be passed to the condition node, they need to be joined to confirm that both values are present when the comparison is made. The mocked sensors that represented the entities in this use case were given the values that trigger the rule, resulting in the rule being sent for translation. The format of the rule after it was prepared for the translation can be found in Appendix A.3.2. The AST formed from the parse tree can be seen in

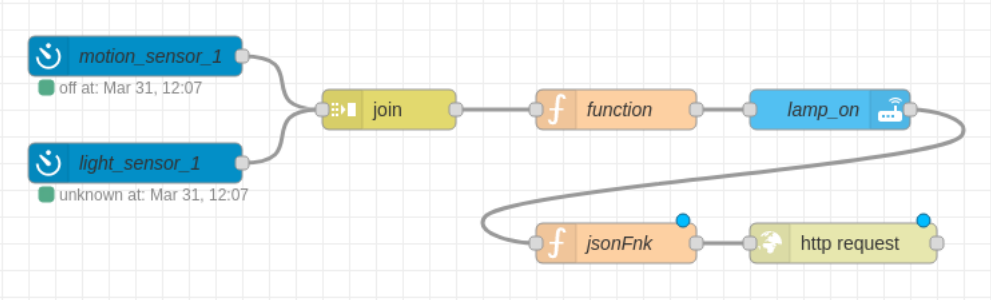

Figure 9.3: The second use case implemented in Node-RED.

Figure 9.4. Since the use case has only one if-clause, the AST contains only one $IfBlock$. As with the previous diagram, the $NextWires$ of the $IfBlock$ contain the id of the "jsonFnk" node. In contrast to the previous one, in this AST there are multiple $Cases$ in the $IfBlock$. This is because

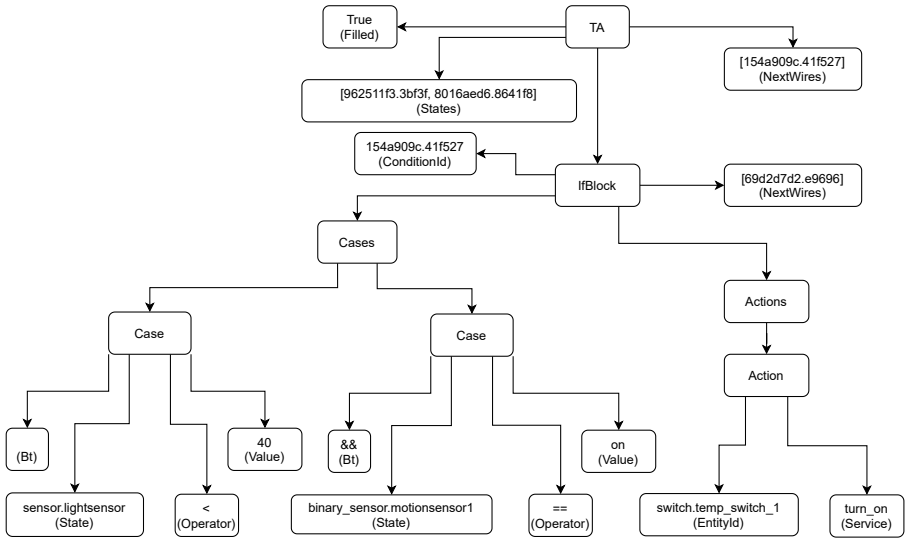

Figure 9.4: The second use case's AST.

the if-clause in the condition has two comparisons. With the AST done, it was proceeded to the next step, which was assembling the text. The result of this was the following:

- When lightsensor detects a value that is less than 40 % and motionsensor1 detects motion, then Lamp will be turned on.

Similar to the previous use case, the result of the translation is quite similar to the explanation of the use case, meaning that the text should be human-readable. In this use case, there are multiple sensors. It represents all rules that have multiple sensors and an action afterward, meaning that they have multiple triggers.

The manner in which the last use case was implemented with Node-RED is presented on Figure 9.5. As it can be seen in the figure, this is the most complex of the use cases. The use case can be explained in the following way:

- When there is motion in the room and the temperature is above 24 °C and the window is opened, then the window should be closed and the HVAC system should be turned on. After this, if the light in the room is above 55 % then the blinds should be closed.

The first part of the rule is implemented like it was done for the previous use case, the only difference being that there are multiple action nodes after the condition. The condition in this use case is the "function_node", and the actions are the nodes called "window_close" and "hvac_on".

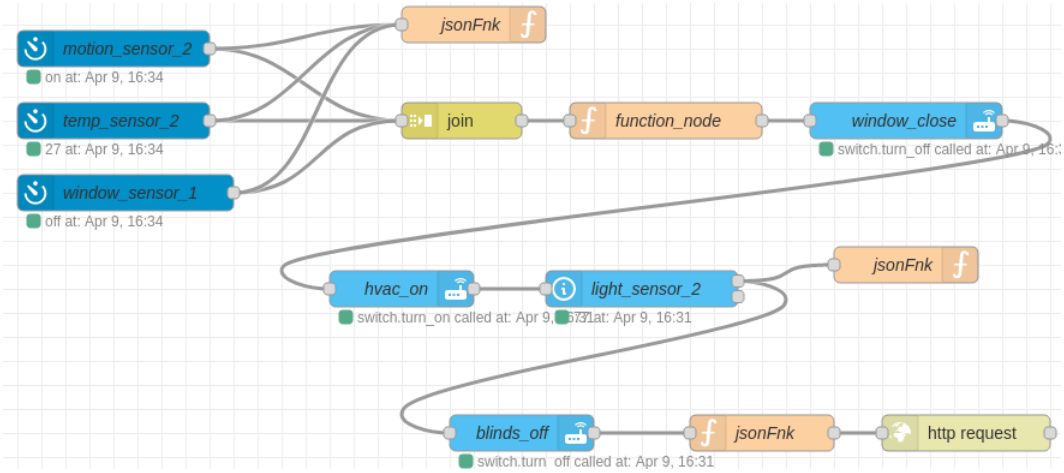

Figure 9.5: The third use case implemented in Node-RED.

Afterwards, however, there is a second TA component nested in the first one. This means that when the condition of the first component is fulfilled there is a second one. It contains a state node("light_sensor_2") and an action node("blinds_off"). The state node has an inner condition as the condition only has one comparison. Because of the nested TA component, additional helper function nodes were added. Those helper functions add the values of the state list to a global flow variable. This is done because when there are multiple sequentially placed state nodes, e.g. the nodes "temp_sensor_2" and "light_sensor_2" in Figure 9.5, the value in the second TA overwrites the ones in the first. If there were no helper function nodes, the request would contain only the values of the last state node in the sequence("light_sensor_2"). They were named "jsonFnk" as the node before the node that sends the HTTP request. The reason for this is that nodes named "jsonFnk" are removed from the rule when the rule is prepared for the translation.

The JSON of the rule, after the rule was prepared for the parsing, can be found in Appendix A.3.3. The parse tree resulting from the parsing was used to build the AST seen in Figure 9.6. This diagram was a lot more complex because it has a nested $IfBlock$. The first $IfBlock$ was similar to the previous diagrams. The only difference was the increase in $Cases$ and $Actions$. The second $IfBlock$ is the interesting part in this use case because it makes the rule more complicated. The $NextWires$ of the first $IfBlock$ contain all of the ids of the "jsonFnk" nodes. The second one contains the ids of the "jsonFnk" nodes that come after it. From the AST the following text was formed:

- When tempsensor2 detects a value that is higher than 24 °C and motionsensor2 detects motion and windowsensor1 detects window is open, then Window Actuator 1 will be closed and HVAC System will be turned on. Afterward, if lightsensor2 detects a value that is higher than 55 %, then Blinds will be closed.

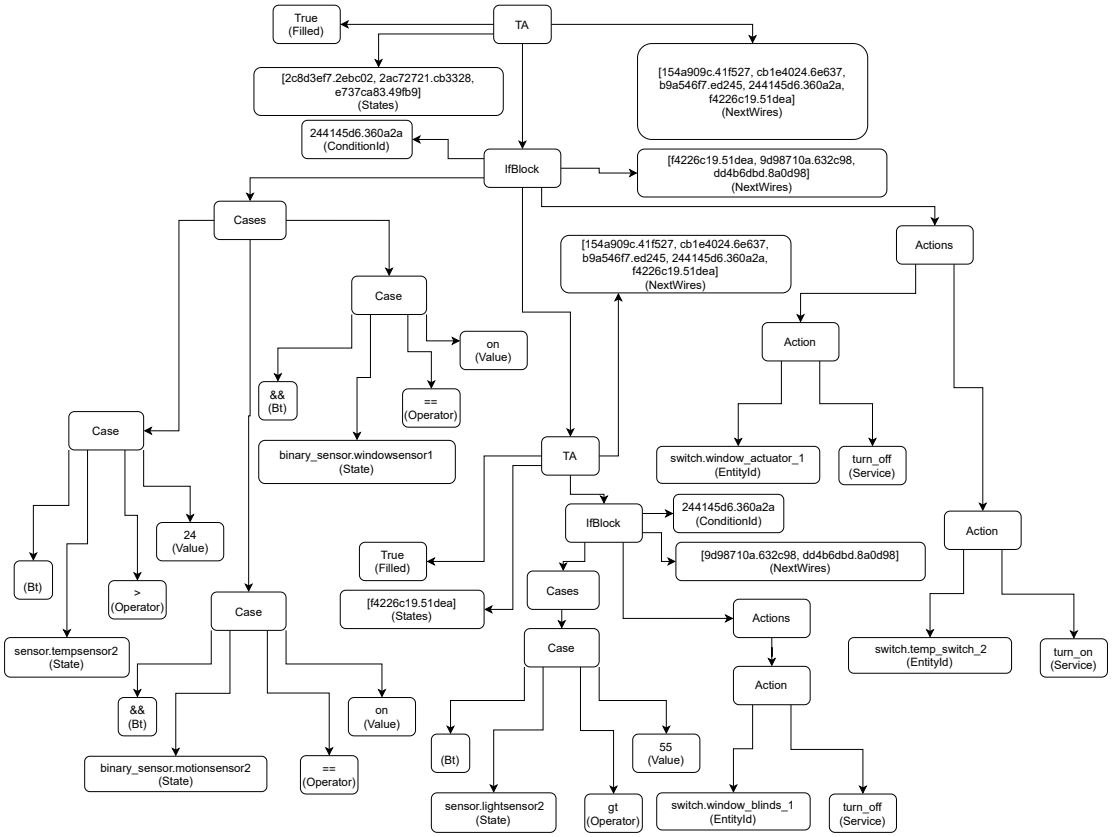

Figure 9.6: The third use case's AST.

The text has the same meaning as the initial explanation. The last use case represents more complex rules that contain more than one TA components in a sequence. It also has a TA component with multiple triggers and actions. This means that the translation component can also translate more complex rule sequences in an adequate way.

9.2 Evaluation of Collection

For the purpose of evaluating the collection of the rule feedback, each of the rules created from the use cases mentioned in the previous section was assigned to a different Area. This was done by assigning a sensor from each of the rules to a mock device. The device was then added to one of three available Areas.

The first case was set in the Area "Living Room", the second in the "Kitchen" and the third in the "Bedroom". Figure 9.7 shows how the rules of the use cases were shown in the UI. As the rules were set in different Areas when their assigned room was chosen, only the rule that was in that room was shown.

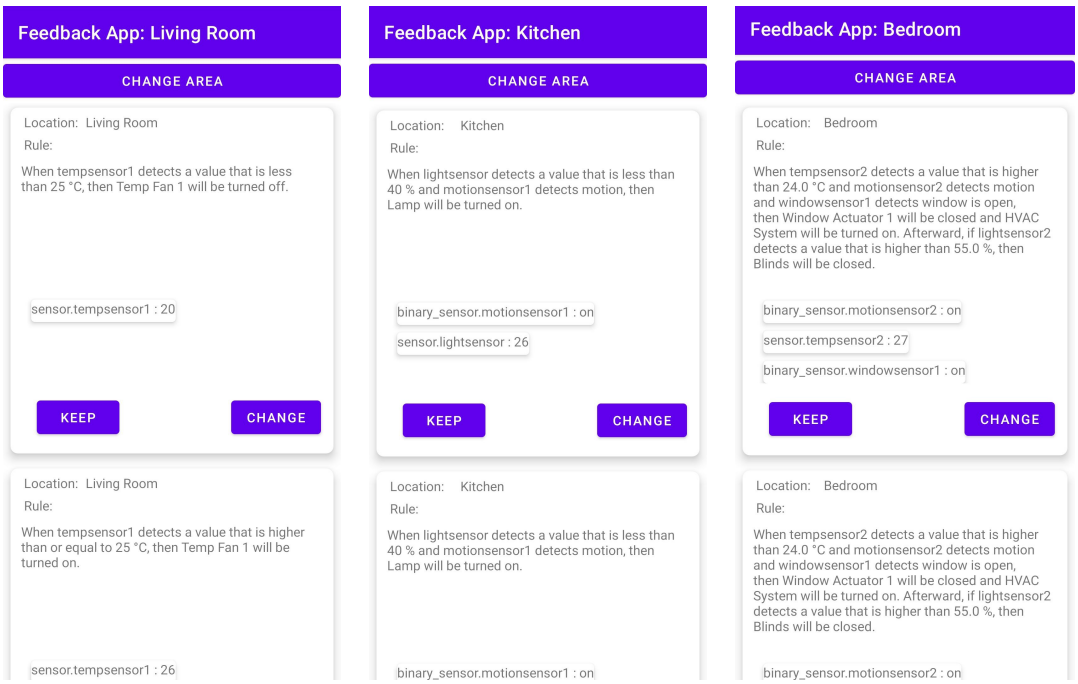

Figure 9.7: The rules of the use cases shown in the UI.

Afterwards, the values of each of the rules were positively and negatively reviewed. When the review was positive, the exact same values were sent back to the database. When they were reviewed negatively, for each of the rules, each of the possible values was changed. After this was done from the same device multiple times, the values were changed identically to the last given feedback. As previously mentioned users are identified by their devices. When feedback was given from multiple devices in the same Area, the values that were updated were the means(or modes) of the values given in the feedbacks of the multiple users.

9.3 Evaluation of Research Questions

This thesis had two research questions that were defined in Chapter 1. In this section, each research question's solution is discussed separately. The following paragraphs present the solutions.

***RQ*$_1$**: The first research question deals with how home automation rules can be presented in a human-readable format to the user. The process of translating a rule is presented in Chapter 6 and Chapter 7. To answer the question, first, a model for an automation rule was determined. Using the model, a grammar was created and given to a parser to parse the rule to a parse

tree. The parse tree was used to build an AST which contained only the aspects of a rule that were relevant to the text formation. The AST was then traversed, and by adding connecting words, sentences were built from it.

***RQ*$_2$**: The second research question is concerned with the topic of how users' feedback is collected after they have been provided a human-readable representation of a triggered rule. Chapter 8 presents the answer to this question. To be able to collect the feedback, the users first need to be able to give their feedback. For this, a mobile application was designed where the users receive the rule after it has been triggered. When the users submit their feedback, it is sent to the back-end. There the mean values from the users' feedback were calculated. These values were then applied to the automation rules.

9.4 Limitations

This section talks about the limitations of the concepts presented in this thesis. A limitation that spans the whole project was the fact that because of the SARS-CoV-2 restrictions, limiting the access to the university, it was not possible to use real sensors and actuators. This means that the system has only been tested with sensors and actuators that were mocked for it. It is possible that using it with real devices may require some alterations to the system.

9.4.1 Rule Implementation Limitation

There is a limitation in the way the rule is sent to the translation component. When a rule has nested TA components, then the rule is sent to the translation component when the path of the triggered TA components reaches the node that sends the HTTP request. We take use case three from the first section of this chapter (Figure 9.5) as an example. When the inner condition in the node called "light_sensor_2" is not fulfilled, meaning that the nested TA component is not triggered, then the rule is not sent to the translation component.

9.4.2 Translation Component Limitations

The translation component has a few limitations, which will be discussed in this subsection. First, starting with the extent to which the component supports the nodes part of Node-RED. Since the idea of this thesis is to provide a proof of concept and because of time restrictions, not all of the Node-RED nodes were supported by the component. Only the nodes that were needed for implementing the use cases mentioned in the previous section were added.

Another limitation of the translation connected with Node-RED is the assumption about the structure of a function node. Normally the function node can contain any JavaScript code. For the translation component, it is, however, assumed that the code of the node has a specific form. This was done again because of time restrictions and it not being the main focus of the thesis. Furthermore, having a complex JavaScript code within a node would have made the rule untranslatable. An additional limitation related to functions is that they cannot have mixed conditions. This means that there cannot be an "and" and an "or" in the same condition.

There is a limitation concerning the flow nodes. Since they do not add specific meaning to a rule, they were generalized too broadly. Because of time restrictions, they were not studied enough. With the defined rule model and the small number of use cases, it was hard to envision where such nodes could be placed, except the join node after the state nodes. Therefore in the grammar, their place is limited to the position in front of a condition node. To allow for easier changes when more roles can be determined for such nodes, the AST does not limit their position.

A further limitation is the vocabulary concerning the sensors and actuators. Because of time restrictions, only vocabulary relevant to the types of sensors that were part of the use cases was supported by the part of the translation component that builds the text. If the sensor has values that use very specific terms, they are not currently supported.

How a rule is translated is closely dependent on the rule model on which the parsing and then the building of the text are based. The grammar that was designed does not fully tolerate parallel conditions. The only parallel conditions that are possible are if- and else-if-clauses part of the same condition node that overlap when the rule is triggered. The structure of the AST does not support parallel condition nodes at all. Therefore, they are not supported by the system.

An additional weakness of the translation component is the nesting of conditions. It supports nesting without any problem when the conditions in the cases do not intersect. If the conditions of one condition node intersect, this means that there could be multiple paths that are triggered. The result of this makes explaining the rule a lot more complex. The resulting text after the translation of such a rule might prove to be unclear and confusing.

9.4.3 User Location Limitations

The manner in which the location of a user is determined has limitations. In the concept proposed by this thesis, the location of the users is determined by asking them for it. The problem with this is that it cannot be guaranteed that the location that the users choose is their real location. Since the location is used to regulate which user can give a feedback to a rule, having a user that gives a false location can lead to using feedback that is not relevant.

When researching how to locate the device a user uses for giving his feedback there were multiple options. HA provides two methods for detecting the presence of a user [Homf]. Users can be tracked by stationary trackers, tracking their devices using Bluetooth or their connection to the router. The problem with such trackers is that it is hard to establish rooms. The router tracker can track if the user is connected to it but cannot determine where exactly he is in the house. With the Bluetooth tracker, the location can be narrowed down slightly, but it still cannot establish in which exact room the user is.

The other method for discovering the user's locations is using Global Positioning System (GPS) trackers. They can find the exact location of the user's device. The problem with such trackers is that GPS coordinates track in a two-dimensional manner. This means that if there are multiple stories in the building the user is in, it cannot be determined on which floor the room is. Therefore, the exact room cannot be determined. As a result, no reliable solution for automatically discovering if the user was in the same room as the sensors was found. Additionally, localisation is a complex topic that is not in the scope of the research of this thesis. Because of this, asking the user for the information was the method that was chosen.

9.4.4 UI Component Limitations

Because of time restrictions and it not being too relevant to the scope of this thesis the UI components has several limitations. The first is that the UI does not scale well with different devices. Furthermore, there are multiple UI bugs, e.g. text not resizing correctly or the keyboard appearing over the text field to which the text is inputted. Additionally, for a rule to be shown when it is triggered the application has to be opened.

9.4.5 Rule Updating Component Limitations

The component that updates the automation rules using the user's feedback is also dependent on the assumption that the function node has a specific form. This results from the fact that it also uses parsing for the JavaScript function inside the function node. The reasons for this is the same as with the limitation of the translation component concerning the structure of the function node.

There is another limitation to the updating process, however, it is a limitation of Node-RED. It is the fact that when a rule is updated and send back to Node-RED, the changes need approval before they can be added to the rules. There was no way to bypass this since it is a part of Node-RED that cannot be changed.

10 Conclusion and Future Work

The purpose of this thesis was to research the topic of dynamic automation rules and how they could be updated using explicit user feedback. The two main objectives of the research were summarized in the two research questions. The first question concerned the form in which the user received rules, in particular how they could be made human-readable. To answer it, first, a rule model was established. Literature research showed that the TA model can support most of the users' desires for automation rules. The model was extended to support more complex rules. Based on the rule model, a grammar was created and then used to build a parser for the rules. The parser parsed the rules and created parse trees from them. After a parse tree for a rule was created, it was traversed, and from parts of it that were relevant to the textual representation, an AST was created. Because of the characteristics of ASTs, it was possible to use the tree directly to build a text. Additional words and phrases were added to fill the gaps between the nodes of the tree and assemble a coherent text. This way the text of each rule was assembled. The explained process provided the answer to the first research question.

By having the textual representation of a rule, the users were now able to understand a triggered rule so they could provide their feedback. Feedback collection is also the topic of the second research question, namely how the feedback of the users can be collected. For this purpose, a mobile application was created. The application served as the UI where the users could see the triggered rules and provide feedback. The provided feedback was then sent to a component that updates the rules. Additionally, before the rule that was reviewed was updated, the user's feedback was stored. Using all user feedbacks for the given rule, the values that are used for updating were calculated. The values were computed by taking the average values from the feedbacks of all users for the given rule. This process provided a concept that answered the second research question.

The translation component was verified by implementing multiple use cases in Node-RED as rules and then translating them. The whole translation process was examined using the values from the use cases. The textual representations of the rules created from the use cases were observed and compared with their initial description. Afterwards, the use cases were also used to test the component that collects and updates the rules. Feedback was given for all rules and its effects on the defined rules was observed.

The results of the evaluation showed that the proposed system can successfully translate the provided use cases. The resulting text was also very similar to the initial description of the use cases. Furthermore, the proposed feedback collection method allowed to change the rules, when providing feedback to the use case rules in the mobile application. The changes were correctly added to the defined use case rules in Node-RED.

The concepts presented in this thesis support most simple rules that do not contain parallel or nested TA components. With the increase of the complexity of the rule, it becomes harder for the translating component to keep the text understandable. Furthermore, it is important that the rule structure complies with the rule model that was used for the translation.

During the research for this thesis, several lessons were learned. The most important one is that Node-RED does not allow the automatic update of rules. Changes have to be manually merged before they are applied. Additionally, when the defined rules are very complex, even if they are translated to a textual form, the text can still be incomprehensible to the users. This results from the complex structure of the rule and not the translation.

Many parts of this thesis' results can be used for future research. One thing that can surely be improved is using the implemented concept with real sensors and actuators and possibly make a study with real people using the system. This would provide more validation for the created system. Additionally, it would require more of Node-RED's node types to be implemented in the system.

A further improvement would be enabling the function node in Node-RED to have an arbitrary form since, in this thesis, there were multiple assumptions about its construction. It would be interesting to research the formation of human-readable rules when there are nodes that can themselves have a quite complex structure and meaning.

Another possible extension that can be done is using machine learning on the feedback so the rule can be optimized. This can make the rules more personal, as the concept implemented in this thesis already tracks which user sends which feedback.

An additional topic for research would be this solution's limitations. First, research could be done on the rule model, determining if there is a model that generalizes rules better. Since with the solution of this thesis there is the possibility that there can be rules which do not comply with the defined model, it would be interesting to examine if there is a model that both generalizes better and can be used to translate to a human-readable format.

Bibliography

[BHSB17] J. Buhl, M. Hasselkuß, P. Suski, H. Berg. "Automating Behavior? An Experimental Living Lab Study on the Effect of Smart Home Systems and Traffic Light Feedback on Heating Energy Consumption". In: *British Journal of Applied Science and Technology* 22 (July 2017), pp. 1–1834414. DOI: 10.9734/CJAST/2017/34414 (cit. on p. 9).

[BLM+11] A. B. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, C. Dixon. "Home Automation in the Wild: Challenges and Opportunities". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, pp. 2115–2124. ISBN: 9781450302289. DOI: 10.1145/1978942.1979249. URL: https://doi.org/10.1145/1978942.1979249 (cit. on pp. 1, 9).

[BV15] B. R. Barricelli, S. Valtolina. "Designing for end-user development in the internet of things". In: *International symposium on end user development*. Springer. 2015, pp. 9–24 (cit. on p. 16).

[BXL+18] L. Bu, W. Xiong, C.-J. M. Liang, S. Han, D. Zhang, S. Lin, X. Li. "Systematically Ensuring the Confidence of Real-Time Home Automation IoT Systems". In: *ACM Trans. Cyber-Phys. Syst.* 2.3 (June 2018). ISSN: 2378-962X. DOI: 10.1145/3185501. URL: https://doi.org/10.1145/3185501 (cit. on p. 9).

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier. "Clone detection using abstract syntax trees". In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 1998, pp. 368–377. DOI: 10.1109/ICSM.1998.738528 (cit. on p. 25).

[Cha87] N. P. Chapman. *LR parsing: theory and practice*. CUP Archive, 1987 (cit. on p. 11).

[CP15] M. Collotta, G. Pau. "Bluetooth for Internet of Things: A fuzzy approach to improve power management in smart homes". In: *Computers Electrical Engineering* 44 (2015), pp. 137–152. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2015.01.005. URL: http://www.sciencedirect.com/science/article/pii/S0045790615000117 (cit. on p. 1).

[Dag] S. Dague. *Why can't we have the Internet of Nice Things? A home automation primer*. URL: https://opensource.com/article/17/7/home-automation-primer (visited on 04/26/2021) (cit. on p. 5).

[DMC09] Z. Drey, J. Mercadal, C. Consel. "A Taxonomy-Driven Approach to Visually Prototyping Pervasive Computing Applications". In: *Domain-Specific Languages*. Ed. by W. M. Taha. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 78–99. ISBN: 978-3-642-03034-5 (cit. on p. 9).

[DS17] R. Dhall, V. Solanki. "An IoT Based Predictive Connected Car Maintenance." In: *International Journal of Interactive Multimedia & Artificial Intelligence* 4.3 (2017) (cit. on p. 7).

[DSSK06] A. Dey, T. Sohn, S. Streng, J. Kodama. "iCAP: Interactive Prototyping of Context-Aware Applications". In: May 2006, pp. 254–271. ISBN: 978-3-540-33894-9. DOI: 10.1007/11748625_16 (cit. on p. 15).

[Ecla] Eclipse Fundation. *Paho*. URL: https://www.eclipse.org/paho/ (visited on 04/26/2021) (cit. on p. 7).

[Eclb] Eclipse Mosquitto. *Eclipse Mosquitto*. URL: https://mosquitto.org/ (visited on 04/26/2021) (cit. on p. 7).

[GIT20] S. Graef, D. Ivanova, A. Tiessen. *Analysis of Open-Source Home Automation Systems*. 2020 (cit. on pp. 5, 13).

[GJ91] D. Grune, C. J. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood Ltd, 1991. ISBN: 0136514316,9780136514312 (cit. on p. 21).

[GMPS17] G. Ghiani, M. Manca, F. Paternò, C. Santoro. "Personalization of context-dependent applications through trigger-action rules". In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 24.2 (2017), pp. 1–33 (cit. on p. 16).

[GY16] V. S. Gunge, P. S. Yalagi. "Smart home automation: a literature review". In: *International Journal of Computer Applications* 975 (2016), p. 8887 (cit. on p. 5).

[Homa] Home Assistant Inc. *Automating Home Assistant*. URL: https://www.home-assistant.io/getting-started/automation/ (visited on 04/26/2021) (cit. on p. 16).

[Homb] Home Assistant Inc. *Configuration*. URL: https://www.home-assistant.io/integrations/config/ (visited on 04/26/2021) (cit. on p. 5).

[Homc] Home Assistant Inc. *Core Architecture*. URL: https://developers.home-assistant.io/docs/architecture/core (visited on 04/26/2021) (cit. on pp. 5, 6).

[Homd] Home Assistant Inc. *Home Assistant Frontend*. URL: https://developers.home-assistant.io/docs/frontend (visited on 04/26/2021) (cit. on p. 5).

[Home] Home Assistant Inc. *Integrations*. URL: https://www.home-assistant.io/integrations (visited on 04/26/2021) (cit. on p. 5).

[Homf] Home Assistant Inc. *Person*. URL: https://www.home-assistant.io/integrations/person/ (visited on 04/26/2021) (cit. on p. 50).

[HSK13] C. C. W. Ham, S. P. N. Singh, M. Kearney. "Learning-based model predictive control and user feedback in home automation". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 2718–2724. DOI: 10.1109/IROS.2013.6696740 (cit. on p. 9).

[HTS08] U. Hunkeler, H. L. Truong, A. Stanford-Clark. "MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks". In: *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*. 2008, pp. 791–798. DOI: 10.1109/COMSWA.2008.4554519 (cit. on p. 6).

[Infa] InfluxData Inc. *InfluxDB glossary*. URL: https://docs.influxdata.com/influxdb/v1.8/concepts/glossary/ (visited on 04/26/2021) (cit. on p. 8).

[Infb] InfluxData Inc. *InfluxDB is a time series platform*. URL: https://www.influxdata.com/ (visited on 04/26/2021) (cit. on p. 8).

[Infc] InfluxData Inc. *InfluxDB line protocol tutorial*. URL: https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_tutorial/ (visited on 04/26/2021) (cit. on p. 8).

[Infd] InfluxData Inc. *Time series database (TSDB) explained*. URL: https://www.influxdata.com/time-series-database/ (visited on 04/26/2021) (cit. on p. 8).

[JOC+17] T. Jakobi, C. Ogonowski, N. Castelli, G. Stevens, V. Wulf. "The Catch(Es) with Smart Home: Experiences of a Living Lab Field Study". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: Association for Computing Machinery, 2017, pp. 1620–1633. ISBN: 9781450346559. DOI: 10.1145/3025453.3025799. URL: https://doi.org/10.1145/3025453.3025799 (cit. on p. 1).

[KFBL16] A. B. Karami, A. Fleury, J. Boonaert, S. Lecoeuche. "User in the Loop: Adaptive Smart Homes Exploiting User Feedback—State of the Art and Future Directions". In: *Information* 7.2 (2016). ISSN: 2078-2489. DOI: 10.3390/info7020035. URL: https://www.mdpi.com/2078-2489/7/2/35 (cit. on p. 9).

[KSZ17] M. R. Khan, S. Sachweh, A. Zuendorf. "Towards User-Centered Assistance in Smart Environments Based on Device Metadata". In: *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing,15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. 2017, pp. 308–311. DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2017.65 (cit. on p. 10).

[KT14] R. Kerr, M. Tondro. "Residential feedback devices and programs: Opportunities for natural gas". In: (Jan. 2014), pp. 43–92 (cit. on p. 9).

[MPC+09] G. Morganti, A. Perdon, G. Conte, D. Scaradozzi, A. Brintrup. "Optimising home automation systems: A comparative study on tabu search and evolutionary algorithms". In: *2009 17th Mediterranean Conference on Control and Automation*. IEEE. 2009, pp. 1044–1049 (cit. on p. 5).

[MQTa] MQTT. *FAC*. URL: https://mqtt.org/faq/ (visited on 04/26/2021) (cit. on p. 6).

[MQTb] MQTT. *MQTT Publish / Subscribe Architecture*. URL: https://mqtt.org/ (visited on 04/26/2021) (cit. on p. 6).

[Nai17] N. Naik. "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP". In: *2017 IEEE International Systems Engineering Symposium (ISSE)*. 2017, pp. 1–7. DOI: 10.1109/SysEng.2017.8088251 (cit. on p. 6).

[NE16] C. Nandi, M. Ernst. "Automatic Trigger Generation for Rule-based Smart Homes". In: Oct. 2016, pp. 97–102. DOI: 10.1145/2993600.2993601 (cit. on p. 9).

[New06] M. W. Newman. "Now We're Cooking: Recipes for End-User Service Composition in the Digital Home". In: 2006 (cit. on p. 1).

[NLC02] Neng-Shiang Liang, Li-Chen Fu, Chao-Lin Wu. "An integrated, flexible, and Internet-based control architecture for home automation system in the Internet era". In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*. Vol. 2. 2002, 1101–1106 vol.2. DOI: 10.1109/ROBOT.2002.1014690 (cit. on p. 9).

[Opea] OpenJS Foundation and Node-RED. *Flows*. URL: https://nodered.org/docs/user-guide/editor/workspace/flows (visited on 04/26/2021) (cit. on p. 7).

[Opeb] OpenJS Foundation and Node-RED. *Node-RED*. URL: https://nodered.org/ (visited on 04/26/2021) (cit. on p. 7).

[Opec] OpenJS Foundation and Node-RED. *Nodes*. URL: https://nodered.org/docs/user-guide/editor/workspace/nodes (visited on 04/26/2021) (cit. on p. 7).

[Oped] OpenJS Foundation and Node-RED. *Palette*. URL: https://nodered.org/docs/user-guide/editor/palette/ (visited on 04/26/2021) (cit. on pp. 13, 14).

[Opee] OpenJS Foundation and Node-RED. *Palette Manager*. URL: https://nodered.org/docs/user-guide/editor/palette/manager (visited on 04/26/2021) (cit. on p. 13).

[Opef] OpenJS Foundation and Node-RED. *Wires*. URL: https://nodered.org/docs/user-guide/editor/workspace/wires (visited on 04/26/2021) (cit. on p. 7).

[Opeg] OpenJS Foundation and Node-RED. *Writing Functions*. URL: https://nodered.org/docs/user-guide/writing-functions (visited on 04/26/2021) (cit. on p. 24).

[ope] openHAB Community and the openHAB Foundation e.V. *Textual Rules*. URL: https://www.openhab.org/docs/configuration/rules-dsl.html#defining-rules (visited on 04/26/2021) (cit. on p. 16).

[Par13] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013 (cit. on p. 8).

[PQ95] T. J. Parr, R. W. Quong. "ANTLR: A predicated-LL (k) parser generator". In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810 (cit. on p. 8).

[RC09] P. Rashidi, D. J. Cook. "Keeping the Resident in the Loop: Adapting the Smart Home to the User". In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 39.5 (2009), pp. 949–959. DOI: 10.1109/TSMCA.2009.2025137 (cit. on p. 9).

[RT17] B. L. Risteska Stojkoska, K. V. Trivodaliev. "A review of Internet of Things for smart home: Challenges and solutions". In: *Journal of Cleaner Production* 140 (2017), pp. 1454–1464. ISSN: 0959-6526. DOI: https://doi.org/10.1016/j.jclepro.2016.10.006. URL: http://www.sciencedirect.com/science/article/pii/S095965261631589X (cit. on p. 1).

[Tera] Terence Parr. *About The ANTLR Parser Generator*. URL: https://www.antlr.org/about.html (visited on 04/26/2021) (cit. on p. 8).

[Terb] Terence Parr. *Download ANTLR*. URL: https://www.antlr.org/download.html (visited on 04/26/2021) (cit. on p. 8).

[Terc] Terence Parr. *What is ANTLR?* URL: https://www.antlr.org/ (visited on 04/26/2021) (cit. on p. 8).

[TYA18] S. Tsuchiya, I. Yamada, K. F. Aoki-Kinoshita. "GlycanFormatConverter: a conversion tool for translating the complexities of glycans". In: *Bioinformatics* 35.14 (Dec. 2018), pp. 2434–2440. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty990. eprint: https://academic.oup.com/bioinformatics/article-pdf/35/14/2434/28913701/bty990.pdf. URL: https://doi.org/10.1093/bioinformatics/bty990 (cit. on p. 10).

[UMPL14] B. Ur, E. McManus, M. Pak Yong Ho, M. L. Littman. “Practical Trigger-Action Programming in the Smart Home”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’14. Toronto, Ontario, Canada: Association for Computing Machinery, 2014, pp. 803–812. ISBN: 9781450324731. DOI: 10.1145/2556288.2557420. URL: https://doi.org/10.1145/2556288.2557420 (cit. on pp. 16, 43).

[Web19] M. Weber. “Context and Interaction in the Internet of Things”. In: (2019) (cit. on p. 5).

[ZWZ+19] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu. “A Novel Neural Source Code Representation Based on Abstract Syntax Tree”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 783–794. DOI: 10.1109/ICSE.2019.00086 (cit. on p. 25).

All links were last followed on April 26, 2021.

A Appendix

A.1 Rule Grammar

```
grammar RuleGrammar;

nodeRule : rule EOF;

ruleSequence: '[' ta (ta|conditionTA)*']';

ta: stateNode+ condition? action+ ;

conditionTA: condition action+;

action: ( (',' '{' actionNodeType '}'));

condition:  (',' '{'flowNode'}')? (',' '{' conditionNodeType '}');

stateNode: ('{' stateNodeType '}' | (',' '{' stateNodeType '}') );

flowNode: (join);

stateNodeType:(eventsState| pollState|currentState);

conditionNodeType:(functionCondition);

actionNodeType:(serviceAction);

eventsState: nodeId ',' stateType ',' dump ',' nodeName ',' dump  ',' entityId ',' dump',' stateCondition ',' dump ',' wires;

pollState: nodeId ',' pollStateType ','dump','nodeName','dump  ',' entityId ',' dump',' stateCondition ',' dump ',' wires;

currentState: nodeId ',' currentStateType ',' dump','nodeName',' dump','currentStateCondition' ,'dump ','entityId','dump ','wires;

functionCondition: nodeId ',' conditionType ','dump','nodeName','  func ',' output',' dump ',' wires;

serviceAction: nodeId',' actionType ',' dump ',' nodeName ',' dump ',' serviceDomain',' service','entityId ','dump','wires ;

join: nodeId(',' innerJoin)+ ',' wires;

```

```
innerJoin:( joinType | dump| nodeName);

stateCondition: haltifState ',' haltiftype ',' haltifcompare ',' output;

currentStateCondition: output','haltifState ',' haltiftype ',' haltifcompare;

func: funcWord ':' value;

dump: pair(',' pair)*;

pair: key ':' value;

key: STRING;

basicJsonObject: '{''}'| '{' dump '}';

array: '[' value (',' value)* ']' | '[' ']';

value: STRING|NUMBER|DOUBLE|'true'|'false'|'null'|array|basicJsonObject;

entityId: entityIdWord ':'  value;

nodeName: nameWord ':' value;

service: serviceWord ':'  value;

serviceDomain: serviceDomainWord ':' value;

haltifState:HALTSTATESTRING ':' value ;

haltiftype: HALTTYPESTRING ':' value;

haltifcompare: HALTCOMPARESTRING ':' ct;

output:OUTPUTKEYSTRING ':' NUMBER;

ct: CT;

nodeId: idkey ':' value;

actionType:  typeword ':' ACTIONTYPEVALUE;

stateType:  typeword ':' STATETYPEVALUE;

pollStateType: typeword ':' POLLSTATETYPEVALUE;

currentStateType: typeword ':' CURRENTSTATETYPEVALUE;

conditionType: typeword ':' CONDITIONTYPEVALUE;

joinType: typeword ':' JOINVALUE;

typeword:  TYPESTRING ;
```

```
90
91  idkey: IDKEYSTRING ;
92
93  nameWord: NAMESTRING ;
94
95  entityIdWord: ENTITYIDSTRING;
96
97  serviceDomainWord: SERVICEDOMAINSTRING;
98
99  serviceWord: SERVICESTRING;
100
101 funcWord: FUNCSTRING;
102
103 wires:wireWord ':'wirevalue;
104
105 wireWord: WIREWORD;
106
107 wirevalue: array;
108
109 fragment GT: ('>'|'gt');
110
111 fragment LT: ('<'|'lt');
112
113 fragment GTE: ('>='|'gte');
114
115 fragment LTE: ('<='|'lte');
116
117 fragment E: ('=='|'is');
118
119 fragment NE:('!='|'is not');
120
121 fragment ESCAPED: '\\' (["\\/bfnrt] | UNICODE);
122
123 fragment UNICODE : 'u' HEX HEX HEX HEX;
124
125 fragment HEX : [0-9a-fA-F];
126
127 NUMBER: INT;
128
129 fragment NR: '"';
130
131 INT: '0' | [1-9] [0-9]*;
132
133 DOUBLE: '-'? INT '.' [0-9]+ EXP?;
134
135 EXP: [Ee] [+\-]? INT;
136
137 TYPESTRING : NR 'type' NR;
138
139 IDKEYSTRING: NR 'id' NR;
140
141 NAMESTRING: NR 'name' NR;
142
```

```
SERVICESTRING: NR 'service' NR;

STATETYPEVALUE: NR 'server-state-changed' NR;

POLLSTATETYPEVALUE: NR 'poll-state' NR;

CURRENTSTATETYPEVALUE: NR 'api-current-state' NR;

ENTITYIDSTRING : NR ('entityidfilter'| 'entityId'|'entity_id') NR;

HALTSTATESTRING : NR ('haltifstate'|'halt_if') NR ;

HALTTYPESTRING: NR 'halt_if_type' NR ;

HALTCOMPARESTRING: NR 'halt_if_compare' NR ;

ACTIONTYPEVALUE: NR 'api-call-service' NR;

CONDITIONTYPEVALUE: NR 'function' NR;

OUTPUTKEYSTRING: NR 'outputs' NR;

FUNCSTRING: NR 'func' NR;

JOINVALUE: NR 'join' NR;

WIREWORD: NR 'wires' NR;

SERVICEDOMAINSTRING: NR 'service_domain' NR;

CT:NR? (GT|LT|GTE|LTE|E|NE) NR?;

STRING: NR (ESCAPED | ~["\\])* NR;

WS: [ \t\n\r]+ -> skip;
```

A.2 Function Grammar

```
grammar FunctionGrammar;

/*
 * Parser Rules
 */
starter: function EOF;

function: ift elseift* elset?;

ift: IF '(' comparison')'STATEMENT;

elseift: ELSE IF '(' comparison')'STATEMENT;

```

```
14 elset: ELSE STATEMENT;
15
16 comparison: case (bt case)*;
17
18 bt:BT;
19
20 case: (value['innerCaseValue']) ct funcValue;
21
22 ct: CT;
23
24 value: VALUE;
25
26 funcValue: VALUE
27
28 innerCaseValue: VALUE;
29
30 /*
31  * Lexer Rules
32  */
33
34 fragment GT: ('>'|'gt');
35
36 fragment LT: ('<'|'lt');
37
38 fragment GTE: ('>='|'gte');
39
40 fragment LTE: ('<='|'lte');
41
42 fragment E: ('=='|'is');
43
44 fragment NE:('!='|'is not');
45
46 fragment LOWERCASE  : [a-z] ;
47
48 fragment UPPERCASE  : [A-Z] ;
49
50 fragment NUM: [0-9];
51
52 fragment CHAR: (LOWERCASE | UPPERCASE|NUM| '-'| '.'|'_');
53
54 fragment ESCAPED: '\\' ([{}\\/bfnrt] | UNICODE);
55
56 fragment UNICODE : 'u' HEX HEX HEX HEX;
57
58 fragment HEX : [0-9a-fA-F];
59
60 IF: 'if';
61
62 ELSE: 'else';
63
64 CT: (GT|LT|GTE|LTE|E|NE) ;
65
66 BT: ('||'|'&&');
```

```
67
68 VALUE: CHAR+ ;
69
70 STATEMENT: '{' (ESCAPED | ~[{}\\])* '}';
71
72 WS: [ \n\t\r]+ -> skip;
```

A.3 Use Case Rules in Node-RED JSON Format

A.3.1 Use Case 1 Node-RED JSON Format


```
1  [
2     {
3        "id":"cb1e4024.6e637",
4        "type":"server-state-changed",
5        "z":"e63925ed.4ac43",
6        "name":"temp_sensor_1",
7        "server":"5b1355ad.60ce64",
8        "version":1,
9        "exposeToHomeAssistant":false,
10       "haConfig":0,
11       "entityidfilter":"sensor.tempsensor1",
12       "entityidfiltertype":"exact",
13       "outputinitially":false,
14       "state_type":"num",
15       "haltifstate":"25",
16       "halt_if_type":"num",
17       "halt_if_compare":"gte",
18       "outputs":2,
19       "output_only_on_state_change":true,
20       "for":0,
21       "forType":"num",
22       "forUnits":"minutes",
23       "ignorePrevStateNull":false,
24       "ignorePrevStateUnknown":false,
25       "ignorePrevStateUnavailable":false,
26       "ignoreCurrentStateUnknown":false,
27       "ignoreCurrentStateUnavailable":false,
28       "x":260,
29       "y":160,
30       "wires":[
31          [
32             "e86c7b0d.27b41"
33          ],
34          [
35             "28c177ad.022568"
36          ]
37       ]
38    },
39    {
40       "id":"e86c7b0d.27b41",
41       "type":"api-call-service",
42       "z":"e63925ed.4ac43",
43       "name":"fan_tun_on",
```

```
        "server":"5b1355ad.60ce64",
        "version":1,
        "debugenabled":false,
        "service_domain":"fan",
        "service":"turn_on",
        "entityId":"temp_fan_1",
        "data":0,
        "dataType":"json",
        "mergecontext":"",
        "output_location":"",
        "output_location_type":"none",
        "mustacheAltTags":false,
        "x":490,
        "y":120,
        "wires":[
           [
              "e1e8f67f.89ec9"
           ]
        ]
     },
     {
        "id":"28c177ad.022568",
        "type":"api-call-service",
        "z":"e63925ed.4ac43",
        "name":"fan_turn_off",
        "server":"5b1355ad.60ce64",
        "version":1,
        "debugenabled":false,
        "service_domain":"fan",
        "service":"turn_off",
        "entityId":"temp_fan_1",
        "data":0,
        "dataType":"json",
        "mergecontext":"",
        "output_location":"",
        "output_location_type":"none",
        "mustacheAltTags":false,
        "x":490,
        "y":200,
        "wires":[
           [
              "e1e8f67f.89ec9"
           ]
        ]
     }
  ]
```


A.3.2 Use Case 2 Node-RED JSON Format


```
  [
     {
        "id":"962511f3.3bf3f",
        "type":"poll-state",
        "z":"76feca1.0773c34",
        "name":"motion_sensor_1",
        "server":"5b1355ad.60ce64",
```

```
        "version":1,
        "exposeToHomeAssistant":false,
        "haConfig":0,
        "updateinterval":"20",
        "updateIntervalUnits":"seconds",
        "outputinitially":false,
        "outputonchanged":false,
        "entity_id":"binary_sensor.motionsensor1",
        "state_type":"str",
        "halt_if":"",
        "halt_if_type":"str",
        "halt_if_compare":"is",
        "outputs":1,
        "x":120,
        "y":120,
        "wires":[
            [
                "f1fb2741.3e9228"
            ]
        ]
    },
    {
        "id":"8016aed6.8641f8",
        "type":"poll-state",
        "z":"76feca1.0773c34",
        "name":"light_sensor_1",
        "server":"5b1355ad.60ce64",
        "version":1,
        "exposeToHomeAssistant":false,
        "haConfig":0,
        "updateinterval":"20",
        "updateIntervalUnits":"seconds",
        "outputinitially":false,
        "outputonchanged":false,
        "entity_id":"sensor.lightsensor",
        "state_type":"str",
        "halt_if":"",
        "halt_if_type":"str",
        "halt_if_compare":"is",
        "outputs":1,
        "x":120,
        "y":200,
        "wires":[
            [
                "f1fb2741.3e9228"
            ]
        ]
    },
    {
        "id":"f1fb2741.3e9228",
        "type":"join",
        "z":"76feca1.0773c34",
        "name":"",
        "mode":"custom",
        "build":"object",
        "property":"payload",
        "propertyType":"msg",
```

```
        "key":"topic",
        "joiner":"\\n",
        "joinerType":"str",
        "accumulate":false,
        "timeout":"",
        "count":"2",
        "reduceRight":false,
        "reduceExp":"",
        "reduceInit":"",
        "reduceInitType":"",
        "reduceFixup":"",
        "x":310,
        "y":160,
        "wires":[
           [
              "154a909c.41f527"
           ]
        ]
     },
     {
        "id":"154a909c.41f527",
        "type":"function",
        "z":"76feca1.0773c34",
        "name":"function_node",
        "func":"if(msg.payload[sensor.lightsensor]<40&&msg.payload[binary_sensor.motionsensor1]==on){
return msg;}",
        "outputs":1,
        "noerr":0,
        "initialize":"",
        "finalize":"",
        "x":500,
        "y":160,
        "wires":[
           [
              "f4f54c14.e34d6"
           ]
        ]
     },
     {
        "id":"f4f54c14.e34d6",
        "type":"api-call-service",
        "z":"76feca1.0773c34",
        "name":"lamp_on",
        "server":"5b1355ad.60ce64",
        "version":1,
        "debugenabled":false,
        "service_domain":"switch",
        "service":"turn_on",
        "entityId":"temp_switch_1",
        "data":0,
        "dataType":"json",
        "mergecontext":"",
        "output_location":"",
        "output_location_type":"none",
        "mustacheAltTags":false,
        "x":680,
        "y":140,
```

```
        "wires":[
           [
              "69d2d7d2.e9696"
           ]
        ]
     }
]
```

A.3.3 Use Case 3 Node-RED JSON Format

```
[
   {
      "id":"2c8d3ef7.2ebc02",
      "type":"poll-state",
      "z":"be13a4f2.872a1",
      "name":"motion_sensor_2",
      "server":"5b1355ad.60ce64",
      "version":1,
      "exposeToHomeAssistant":false,
      "haConfig":0,
      "updateinterval":"10",
      "updateIntervalUnits":"seconds",
      "outputinitially":false,
      "outputonchanged":false,
      "entity_id":"binary_sensor.motionsensor2",
      "state_type":"str",
      "halt_if":"",
      "halt_if_type":"str",
      "halt_if_compare":"is",
      "outputs":1,
      "x":160,
      "y":140,
      "wires":[
         [
            "714de44a.49375c",
            "b9a546f7.ed245"
         ]
      ]
   },
   {
      "id":"2ac72721.cb3328",
      "type":"poll-state",
      "z":"be13a4f2.872a1",
      "name":"temp_sensor_2",
      "server":"5b1355ad.60ce64",
      "version":1,
      "exposeToHomeAssistant":false,
      "haConfig":0,
      "updateinterval":"10",
      "updateIntervalUnits":"seconds",
      "outputinitially":false,
      "outputonchanged":false,
      "entity_id":"sensor.tempsensor2",
      "state_type":"str",
      "halt_if":"",
      "halt_if_type":"str",
```

```
        "halt_if_compare":"is",
        "outputs":1,
        "x":160,
        "y":200,
        "wires":[
            [
                "714de44a.49375c",
                "b9a546f7.ed245"
            ]
        ]
    },
    {
        "id":"e737ca83.49fb9",
        "type":"poll-state",
        "z":"be13a4f2.872a1",
        "name":"window_sensor_1",
        "server":"5b1355ad.60ce64",
        "version":1,
        "exposeToHomeAssistant":false,
        "haConfig":0,
        "updateinterval":"10",
        "updateIntervalUnits":"seconds",
        "outputinitially":false,
        "outputonchanged":false,
        "entity_id":"binary_sensor.windowsensor1",
        "state_type":"str",
        "halt_if":"",
        "halt_if_type":"str",
        "halt_if_compare":"is",
        "outputs":1,
        "x":170,
        "y":260,
        "wires":[
            [
                "714de44a.49375c",
                "b9a546f7.ed245"
            ]
        ]
    },
    {
        "id":"714de44a.49375c",
        "type":"join",
        "z":"be13a4f2.872a1",
        "name":"",
        "mode":"custom",
        "build":"object",
        "property":"payload",
        "propertyType":"msg",
        "key":"topic",
        "joiner":"\\n",
        "joinerType":"str",
        "accumulate":false,
        "timeout":"",
        "count":"3",
        "reduceRight":false,
        "reduceExp":"",
        "reduceInit":"",
```

```
        "reduceInitType":"",
        "reduceFixup":"",
        "x":450,
        "y":200,
        "wires":[
           [
              "244145d6.360a2a"
           ]
        ]
     },
     {
        "id":"244145d6.360a2a",
        "type":"function",
        "z":"be13a4f2.872a1",
        "name":"function_node",
        "func":"if(msg.payload[sensor.tempsensor2]>24&& msg.payload[binary_sensor.motionsensor2]==on&&
msg.payload[binary_sensor.windowsensor1]==on){    return msg;}",
        "outputs":1,
        "noerr":0,
        "initialize":"",
        "finalize":"",
        "x":620,
        "y":200,
        "wires":[
           [
              "363203ad.59b19c"
           ]
        ]
     },
     {
        "id":"363203ad.59b19c",
        "type":"api-call-service",
        "z":"be13a4f2.872a1",
        "name":"window_close",
        "server":"5b1355ad.60ce64",
        "version":1,
        "debugenabled":false,
        "service_domain":"switch",
        "service":"turn_off",
        "entityId":"window_actuator_1",
        "data":0,
        "dataType":"json",
        "mergecontext":"",
        "output_location":"",
        "output_location_type":"none",
        "mustacheAltTags":false,
        "x":820,
        "y":200,
        "wires":[
           [
              "4f02f179.8f9c2"
           ]
        ]
     },
     {
        "id":"4f02f179.8f9c2",
        "type":"api-call-service",
```

```
        "z":"be13a4f2.872a1",
        "name":"hvac_on",
        "server":"5b1355ad.60ce64",
        "version":1,
        "debugenabled":false,
        "service_domain":"switch",
        "service":"turn_on",
        "entityId":"temp_switch_2",
        "data":0,
        "dataType":"json",
        "mergecontext":"",
        "output_location":"",
        "output_location_type":"none",
        "mustacheAltTags":false,
        "x":400,
        "y":340,
        "wires":[
            [
                "f4226c19.51dea"
            ]
        ]
    },
    {
        "id":"f4226c19.51dea",
        "type":"api-current-state",
        "z":"be13a4f2.872a1",
        "name":"light_sensor_2",
        "server":"5b1355ad.60ce64",
        "version":1,
        "outputs":2,
        "halt_if":"55",
        "halt_if_type":"num",
        "halt_if_compare":"gt",
        "override_topic":false,
        "entity_id":"sensor.lightsensor2",
        "state_type":"num",
        "state_location":"payload",
        "override_payload":"msg",
        "entity_location":"data",
        "override_data":"msg",
        "blockInputOverrides":false,
        "x":600,
        "y":340,
        "wires":[
            [
                "c1a8a08d.e1d8b8",
                "9d98710a.632c98"
            ],
            [

            ]
        ]
    },
    {
        "id":"c1a8a08d.e1d8b8",
        "type":"api-call-service",
        "z":"be13a4f2.872a1",
```

```
        "name":"blinds_off",
        "server":"5b1355ad.60ce64",
        "version":1,
        "debugenabled":false,
        "service_domain":"switch",
        "service":"turn_off",
        "entityId":"window_blinds_1",
        "data":0,
        "dataType":"json",
        "mergecontext":"",
        "output_location":"",
        "output_location_type":"none",
        "mustacheAltTags":false,
        "x":500,
        "y":460,
        "wires":[
            [
                "dd4b6dbd.8a0d98"
            ]
        ]
    }
]
```

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 03.05.2021

place, date, signature