

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Graph-based Approach for Detecting Misconfigurations in Smart Home Automation Rules

Pedro Varandas da Costa Azevedo da Ponte



FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática e Computação

Supervisor: André Restivo

Second Supervisor: João Pedro Dias

July 28, 2023

A Graph-based Approach for Detecting Misconfigurations in Smart Home Automation Rules

Pedro Varandas da Costa Azevedo da Ponte

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee

President: Prof. Pedro Nuno Ferreira da Rosa da Cruz Diniz

External Examiner: Prof. Ângelo Manuel Rego e Silva Martins

Supervisor: Prof. André Monteiro de Oliveira Restivo

July 28, 2023

Abstract

The Internet of Things (IoT) is a digital interconnection of everyday physical devices — “things” — with the Internet. These hardware devices are embedded with sensors, software, and technologies and can connect, share and interact with their users and other intelligent objects. Nowadays, smart devices are available to everyone and, therefore, are used to build smart spaces, as they allow day-to-day behaviors to be automated, making life easier for users.

As these devices started to be used by people with different programming experiences, developers felt the need to improve the usability of this equipment to allow every end-user to create valuable integrations on their own easily. In this context, the trigger action model, also known as trigger action programming (TAP), has become widely adopted. If-This-Than-That (IFTTT) is the most outstanding example of such a programming style, which allows the combination of a single trigger (*e.g.*, if it is sunny outside) and an action (*e.g.*, open the curtains) to define a rule.

With the increasing number of devices and rules in the same smart environment, its complexity grows, and unintended behaviors can occur because there is a greater chance that the rules have conflicts without users noticing that. Current solutions can predict potential conflicts that might occur, but most of them are based on simulations or studying the system’s past behavior.

In this work, we proposed a graph-based solution to predict conflicts between rules defined for a smart environment. To do so, we generate all possible combinations of scenarios between the states of existing sensors and actuators in a given smart space and associate each with one of the nodes of a directed graph. Each edge in this graph represents a defined rule for the environment and corresponds to a transition between two different scenarios. Using this strategy, we claim that we can effectively and efficiently detect loops, redundancies, and inconsistencies.

To evaluate this approach, we designed a smart environment containing some devices and then asked ChatGPT to generate a set of rules where there were conflicts of the types we intended to identify. With the generated rulesets, we compare the conflicts identified by our system with those identified by ChatGPT. As this experience left some combinations of conflicting rules unexplored for redundancies and inconsistencies, we conducted a more extensive analysis to know concretely all the situations in which our system can detect these types of conflicts. To assess the efficiency of our solution, we compared a set of metrics obtained when our system was used with and without the implementation of an algorithm that reduced the number of possible scenarios. Through the obtained results, we were able to demonstrate that a graph-based approach to detect conflicts in smart spaces is effectively feasible.

Keywords: IoT, Internet of Things, Smart Spaces, Rule Conflicts, Predictions, Graphs, Connected Components

Resumo

A *Internet of Things* (IoT) é uma interconexão digital de dispositivos físicos quotidianos — “things” — com a Internet. Estes dispositivos de hardware contêm sensores, software e tecnologias e podem-se conectar, partilhar e interagir com os seus utilizadores e outros objetos inteligentes. Hoje em dia, os dispositivos inteligentes estão ao alcance de todos e, por isso, são cada vez mais utilizados para construir espaços inteligentes, pois permitem automatizar comportamentos do dia a dia, facilitando a vida dos utilizadores.

Como estes dispositivos começaram a ser usados por pessoas com diferentes experiências em programação, os desenvolvedores sentiram a necessidade de melhorar a usabilidade destes equipamentos para permitir que cada utilizador final facilmente crie integrações valiosas por conta própria. Nesse contexto, o *trigger action model*, também conhecido como *trigger action programming* (TAP), tornou-se amplamente adotado. If-This-Than-That (IFTTT) é o exemplo mais notável deste estilo de programação, que permite a combinação de um único *trigger* (e.g., se estiver ensolarado lá fora) e uma *action* (e.g., abrir as cortinas) para definir uma regra.

Com o aumento do número de dispositivos e regras num mesmo ambiente inteligente, a sua complexidade aumenta e podem ocorrer comportamentos indesejados, havendo uma maior hipótese das regras terem conflitos entre si sem que os utilizadores se apercebam. As soluções atuais conseguem prever potenciais conflitos que podem ocorrer, mas a maioria delas é baseada em simulações ou no estudo do comportamento passado do sistema.

Neste trabalho, propomos uma solução baseada em grafos para prever conflitos entre regras definidas para um ambiente inteligente. Para isso, geramos todas as combinações possíveis de cenários entre os estados dos sensores e atuadores existentes num determinado espaço inteligente e associamos cada a um dos nós de um grafo direcionado. Cada aresta neste grafo representa uma regra definida para o ambiente e corresponde a uma transição entre dois cenários diferentes. Usando esta estratégia, afirmamos que podemos detectar loops, redundâncias e inconsistências de maneira eficaz e eficiente.

Para avaliar esta abordagem, desenhamos um ambiente inteligente que contém alguns dispositivos e pedimos ao ChatGPT para gerar um conjunto de regras onde existissem conflitos dos tipos que pretendemos identificar. Com os conjuntos de regras gerados, comparamos os conflitos identificados pelo nosso sistema com os identificados pelo ChatGPT. Como essa experiência deixou algumas combinações de regras conflituosas por explorar no caso das redundâncias e inconsistências, realizamos uma análise mais ampla para conhecer concretamente todas as situações em que o nosso sistema é capaz de detectar esses tipos de conflitos. Para avaliar a eficiência de nossa solução, comparamos um conjunto de métricas obtidas quando o nosso sistema foi utilizado com e sem a implementação de um algoritmo que reduzisse o número de cenários possíveis. Através dos resultados obtidos, pudemos demonstrar que uma abordagem baseada em grafos para detectar conflitos em espaços inteligentes é efetivamente viável.

Keywords: IoT, Internet of Things, Smart Spaces, Rule Conflicts, Predictions, Graphs, Connected Components

Agradecimentos

This dissertation is the culmination of five years of Licenciatura/Mestrado em Engenharia Informática e Computação. In this way, I would like to thank all those who have accompanied me over these years and helped to make this goal possible.

First, I would like to thank my supervisors André Restivo and João Pedro Dias for guiding, supporting, and motivating me throughout this journey. They were fundamental to the success of this work. All the knowledge they transmitted and all the suggestions given, as well as the constructive criticism they were making, allowed to raise the quality of this work.

Afterward, I would like to show my deepest gratitude to all my family, especially my parents, my sister, and my grandparents, for making me what I am today and for all the love and motivation shown. They were tireless and always did everything for me to have everything I needed. They were a pillar. Without them, reaching such a high level would not have been possible. I will be eternally grateful. A special word for Avó Guida, who unfortunately could not accompany this journey until the end, but who will undoubtedly be proud.

A special thanks to the group of friends that college gave me — “Turtles” — who accompanied me throughout the course. Together we work, study, suffer, and celebrate our achievements. After all, we were united, and we never left anyone behind. I am very grateful to have known them. Without them, it would not have been the same.

A big thanks to all my padel friends for putting up with me almost every day, for all the games, and for all the long evening meetings we had in that bar. Undoubtedly, these moments made this journey much lighter and more enjoyable and brought me people and teachings that I will certainly take with me for the rest of my life. They are amazing.

I would also like to thank all my other friends for all the advice, the trips, the meetings, and for being a support throughout all these years. Indeed, much of what I am today is also due to you.

Finally, I extend my thanks to all the teachers I had throughout these years of study for all the teachings and advice given and for trying to guide me along the best paths.

Pedro Ponte

*“A good programmer is someone who always
looks both ways before crossing a one-way street.”*

Doug Linder

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	3
1.3	Problem Definition	3
1.4	Goals	4
1.5	Document Structure	4
2	Background	6
2.1	Graphs	6
2.2	Summary	7
3	State of the Art	8
3.1	Methodology	8
3.2	Results	9
3.2.1	Conflict Detection	9
3.2.2	Conflict Resolution	15
3.2.3	Conflict Detection and Resolution	16
3.2.4	Surveys and Reviews	24
3.3	Categorization	28
3.4	Analysis	28
3.4.1	Publication History	29
3.4.2	Result Analysis	29
3.5	Conclusions	30
3.6	Summary	31
4	Problem Statement	32
4.1	Current Issues	32
4.2	Desiderata	33
4.3	Hypothesis	35
4.4	Research Questions	35
4.5	Methodology	37
4.6	Assumptions	38
4.7	Summary	38
5	Implementation	39
5.1	Overview	39
5.2	Implementation Details	40
5.2.1	Rules Definition	40

5.2.2	Get Graph Valid Scenarios and Graph Generation	42
5.2.3	Rules Parsing and Edges Creation	45
5.2.4	Check Periodicity Intersection Between Two Rules	46
5.2.5	Conflict Detection in Rules	47
5.2.6	Export graphs to GEXF format	56
5.3	Known Limitations	58
5.4	Summary	59
6	Evaluation and Validation	60
6.1	Experimental Setup	60
6.1.1	System	61
6.1.2	Carried Out Experiences	61
6.2	Experiments	61
6.2.1	Rule Conflict Detection in a Designed Smart Environment	61
6.2.2	Extensive Analysis of the Detected Conflicts	87
6.3	Effectiveness of the Implementation of Connected Components Algorithm	94
6.4	Conclusions	96
6.5	Validity Threats	96
6.6	Summary	97
7	Conclusions	98
7.1	Conclusions	98
7.2	Challenges	101
7.3	Future Work	101
	References	103
A	dfs() Function Implementation	107
B	List of Smart Devices	109
C	Query to Generate Initial Rule Set	113
D	Initial Set of Rules	119
E	Updated Rule Set	121
F	Redundant Rules Generated	124
G	Inconsistent Rules Generated	125
H	Loop Sets Generated	127
I	Final Set of Rules	129

List of Figures

1.1	Growth of active IoT connections [6]	2
3.1	FORTNIoT algorithm to deduce predictions	11
3.2	CityGuard Structure	18
3.3	Negotiation algorithm proposed by Ahmed <i>et al.</i>	20
3.4	AutoTap overview	21
3.5	The Conflict Engine framework	22
3.6	Framework Architecture	24
3.7	Intelligent environment overview	25
3.8	Conflict handling proposed by Resendes <i>et al.</i>	26
3.9	Conflicts' overview proposed by Pradeep <i>et al.</i>	27
3.10	Publications per year	30
5.1	Resultant vertexes without using connected components algorithm	45
5.2	Resultant vertexes using connected components algorithm	45
5.3	Example of a graph with parallel edges that contains a loop	49
6.1	Graph where rules 7, 14, 32, and 35 are inserted	63
6.2	Number of rules per trigger type	64
6.3	Number of rules per periodicity type	65
6.4	Graphs where rules 39 and 40 are inserted	67
6.5	Graphs where rules 1 and 45 are inserted	71
6.6	Graphs where rules 9 and 47 are inserted	72
6.7	Graphs where rules 26 and 47 are inserted	73
6.8	Graphs where rules 21 and 48 are inserted	74
6.9	Graphs where rules 25 and 52 are inserted	75
6.10	Graphs where rules 21 and 54 are inserted	76
6.11	Graph where rules 31 and 57 are inserted	77
6.12	Graph where rules 59, 60, and 61 are inserted	79
6.13	Graph where rules 4, 62, and 63 are inserted	80
6.14	Graph where rules 1, 59, and 61 are inserted	81
6.15	Graph where rules 64, 65, and 66 are inserted	83
6.16	Edges associated with rules 46 and 61	85
6.17	Number of rules per trigger type between initial and final sets	85
6.18	Number of rules per periodicity type between initial and final sets	86
6.19	Rules' periodicity comparison	88
6.20	Number of Devices vs. Number of Nodes with Exponential Function	96

List of Tables

3.1	Rule conflict types and conditions to judge them	9
3.2	Results Categorization	29
5.1	Comparison between two triggers using the same sensor to detect inconsistencies	52
5.2	Comparison between two triggers using the same sensor to detect redundancies .	56
6.1	Redundancies detected by ChatGPT vs. Redundancies detected by our system . .	66
6.2	Frequency of each pair Rule Triggers / Rules Periodicity in the founded redundant rules	69
6.3	Inconsistencies detected by ChatGPT vs. Inconsistencies detected by our system	70
6.4	Frequency of each pair Rule Triggers / Rules Periodicity in the founded inconsistent rules	78
6.5	Loops detected between all rules in the set	84
6.6	Metrics obtained using our system with connected components algorithm	94
6.7	Metrics obtained using our system without connected components algorithm . . .	95

Abbreviations

ABNA	Agent-Based Negotiation Approach
AI	Artificial Intelligence
AmI	Ambient Intelligence
AP	Action Place
BMC	Bounded Model Checking
CCA	Calculus of Context-Aware Ambient
CCPN	Conditional Colored Petri Net
CDAM	Conflict Detection Algorithm Model
CDR	Conflict Detection and Resolution
CSPR	City Safety and Performance Requirements
CSSA	City State and Service Action
DFS	Depth-First Search
DNF	Disjunctive Normal Form
FI	Feature Interaction
GEXF	Graph Exchange XML Format
GUI	Graphical User Interface
HBAS	Home Building Automation Systems
IFTTT	If-This-Than-That
IoT	Internet of Things
LHA	Linear Hybrid Automata
M2M	Machine-to-Machine
SCPN	Semantic Colored Petri Net
SHS	Smart Home System
SMT	SmartThings
SUMO	Simulation of Urban MObility
TAP	Trigger Action Programming
TP	Trigger Place

Chapter 1

Introduction

1.1	Context	1
1.2	Motivation	3
1.3	Problem Definition	3
1.4	Goals	4
1.5	Document Structure	4

This chapter intends to introduce the project presented in the following chapters, its motivation, looking at the problem that it aims to solve, and its goals. We start with Section 1.1 (p. 1), providing a context to this project in the area of the Internet of Things [14]. In Section 1.2 (p. 3), we explain the problems of the current solutions that motivated us to develop this work. Section 1.3 (p. 3) states the problem we aim to solve based on the previous chapter. In Section 1.4 (p. 4), we present the goals of this dissertation, and in Section 1.5 (p. 4), we describe the document structure and its contents.

1.1 Context

Almost everyone has heard of the Internet of Things (IoT). However, not all of these people will know how to define IoT correctly, how it works, and what are its real applications. Easily and quickly defined, IoT is the set of devices that can connect to the Internet and communicate with each other, sharing data. However, let us now present some more complex definitions of IoT:

- Oracle defines the Internet of Things as “the network of physical objects —“things” — that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the Internet” [30];
- Amazon AWS defines the Internet of Things as “the collective network of connected devices and the technology that facilitates communication between devices and the cloud, as well as between the devices themselves” [4];

- IBM defines the Internet of Things as “the concept of connecting any device (so long as it has an on/off switch) to the Internet and to other connected devices. The IoT is a giant network of connected things and people — all of which collect and share data about the way they are used and about the environment around them” [21];
- Cambridge Dictionary defines the Internet of Things as the “objects with computing devices in them that are able to connect to each other and exchange data using the Internet” [3].

Analyzing all definitions, we notice that the term “device” is common to all. These devices are embedded with sensors, actuators, software, and technologies. These objects can sense and gather data about themselves and the environment surrounding them. These data can then be shared with the other connected devices, contributing to decision-making. The devices are also capable of interacting with their users [17].

During the last few years, IoT devices have become more and more attractive as they have become more accessible with their price dropping. According to Cisco’s Annual Internet Report [6], the number of Machine-to-Machine (M2M) connections will grow nearly 2.4 times between 2018 and 2023, reaching 14.7 thousand million. The global average of devices and connections per capita will grow from 2.4 in 2018 to 3.6 by 2023. Additionally, connected home applications (*e.g.*, home automation, home security, and video surveillance) will represent almost half (48%) of these M2M connections, as we can see in Figure 1.1 (p. 2).

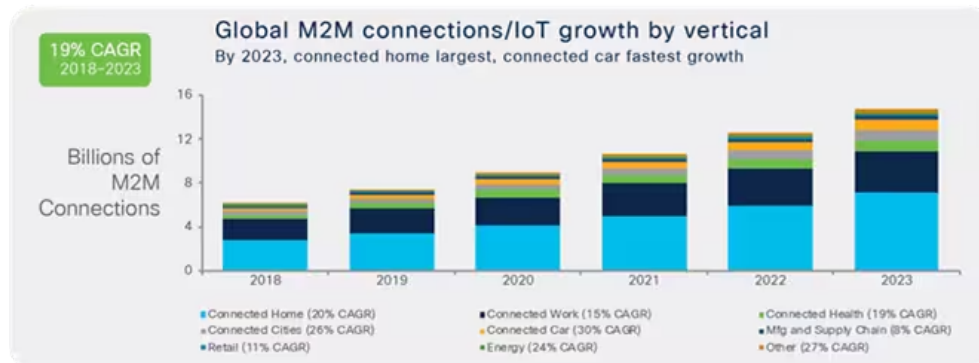


Figure 1.1: Growth of active IoT connections [6]

IoT has applications in several areas, like smart mobility, smart homes/buildings, monitoring, medical and healthcare, and industrial processing, among many others [22].

Focusing on smart homes, words like smart building, automated home, domotics, intelligent house, or smart building are also used to refer to this concept. M. R. Alam defined it in this way:

“A smart home is an application of ubiquitous computing in which the home environment is monitored by ambient intelligence to provide context-aware services and facilitate remote home control.” [1]

Home automation aims to simplify users’ lives, providing comfort, healthcare, and security.

Although these devices make life easier for people and companies, they also bring challenges and problems if misused.

1.2 Motivation

In recent years, we have witnessed massive growth in sales of smart devices. One of the most known use cases for them is its application to home automation. With the emergence of visual programming [36, 42], trigger-action programs (TAP), and voice-assistants [23, 24], the realm of IoT has witnessed a transformative shift, empowering non-programmers to harness its potential. With the growing number of smart devices each person has in their home, the number of trigger-action rules created to get the most out of the devices also grew.

These rules can have an additional complexity for non-programmers, leading to unintended and dangerous behaviors without users realizing the reason. Rules that conflict [33, 34] between them cause these behaviors. A conflict occurs when different rules invoke the same IoT service/device simultaneously, trying to change its state inconsistently, following opposing preferences [28]. Also, a conflict can occur when different rules invoke different IoT services/devices simultaneously, and one action nullifies the impact of the other action [35]. The conflicts can cause infinite loops (when a collection of trigger-action rules is continuously activated without achieving a steady state), inconsistencies (when rules are triggered (almost) at the same time, causing different states for the same device through contradictory actions), or redundancies (when rules are triggered (almost) at the same time, and the action of one is replicated in the action of another), which may lead to unintended behaviors [9]. The unintended behaviors can be more or less problematic: it could just be a light that stays on all day or something more complex that could put the house's safety at risk, like a door that opens when no one is at home.

As the number of rules grows, users have more difficulty understanding all automation behaviors. For a single person, it is already complicated to be able to control all the rules that have been created. Worse will be the case of a shared environment, such as a house, where several people live. It is even more complicated to understand all the existing rules and smart home behaviors and resolve possible conflicts that may arise, as the users do not have full knowledge of all automation.

It is important that users feel they are in control of the smart environment and that it is working as expected. Rule conflicts and unexpected behaviors complicate the situation, as users believe and are convinced that they have configured the system correctly. Then, when some actions do not arise or arise without expecting, users are surprised and do not understand what is happening. These occurrences can demoralize users and give them up creating or using automation.

1.3 Problem Definition

Previous work has already been done predicting possible conflicts and preventing unintended behaviors [7]. However, these solutions based their analysis on debugging past behaviors or simulations.

There is a lack of research that generates and analyzes all possible scenarios from the rules and devices defined for smart spaces, detecting potential loops, redundancies, and inconsistencies effectively. The current systems also fail to present the detected conflicts to the users with a small explanation to facilitate understanding the detected problem.

We argue that a solution capable of detecting/predicting loops, redundancies, and inconsistencies between rules defined for a smart environment, explaining *why* the conflict occurs, will help users fill they have the “power” and control the smart environment [10]. Suppose we manage to equip this system with the ability to build all possible scenarios of combinations between the states of sensors and actuators in that space. In that case, we can predict conflicts in any circumstance. By knowing all possible scenarios, we can provide users with examples of concrete scenarios where the rules will conflict.

1.4 Goals

The main goal of this dissertation is to create a system that, from a set of rules and devices defined for a smart environment, is capable of generating all possible scenarios of combinations between the states of sensors and actuators existing in that space, finding conflicts. The conflicts we want to explore are loops, redundancies, and inconsistencies between rules with triggers that can be executed simultaneously but have different actions in the same actuator.

This system must be efficient, trying to reduce the number of possible scenarios to those strictly necessary so that the conflict detection process does not take too long and uses as little computational resources as possible.

We should be able to explain to end-users *why* a pair of rules are redundant or inconsistent, or *why* a set of rules forms a loop between them.

1.5 Document Structure

This chapter introduces this dissertation’s theme, presenting the context in which the work is inserted and its motivation and defining the problem it aims to solve. Besides this one, this document is composed of six more chapters, structured as follows:

- Chapter 2 (p. 6), **Background**, presents some important concepts to understand the rest of the document;
- Chapter 3 (p. 8), **State of the Art**, gives an overview of the current state of the art on conflict detection and resolution, analyzing the most important approaches;
- Chapter 4 (p. 32), **Problem Statement**, formalizes the problem this work aims to solve by defining a hypothesis. In this chapter, we also define the research questions and the methodology we will use to validate or invalidate the given hypothesis;

- Chapter 5 (p. 39), **Implementation**, describes how we implemented our solution and discusses some decisions we made during this step;
- Chapter 6 (p. 60), **Evaluation and Validation**, presents the evaluation process we made to evaluate our system and answer the Research Questions;
- Chapter 7 (p. 98), **Conclusions**, draws some conclusions about the work done, presents the challenges we faced, and some suggestions for future work.

Chapter 2

Background

In this chapter, we will introduce some concepts necessary to correctly understand what we will present in the rest of the document.

2.1 Graphs

In computer science and mathematics, a graph is a data structure that depicts a collection of linked nodes or vertices and the edges or arcs that connect these nodes. Graphs are used to represent relationships between different objects or entities. More formally, “a graph is a structure containing an object set and its relational data. It models pairwise relations between those objects, often called edges or links. The objects themselves, thus related, in turn being designated as vertices or nodes — used interchangeably throughout this report. It is a pair $G = (V, E)$, where V is the set of vertices and E the set of edges pairing them.” [29].

Graphs can represent different types of relationships and can be applied to any relational data. Therefore, nodes in a graph can represent people, cities, web pages, or any other discrete entity. Edges represent the links or relationships that exist between these nodes. Graphs provide a versatile means of expressing complicated relationships of any scale and may be used to address issues such as route planning, network analysis, and recommendation systems. These structures may be viewed as a mechanism through which significant patterns arise and become more easily detectable [29]. In a graph, two nodes are **neighbors** or **adjacent** if there is an edge between these nodes.

There are several types of graphs. For a correct understanding of what we are going to deal with in this document, it is important to know the following classifications:

- **Undirected or Simple** - Edges have no specific direction and represent symmetric relations between nodes. That is, considering two vertices u and v from an undirected graph, if u is related with v , then v is also related with u .
- **Directed** - Edges have a specific direction, representing one-way relationships between two nodes. An edge only indicates that the starting node u is related to the ending node v . For

the inverse relationship to exist, it is necessary to have an edge with v as starting node and u as the ending node.

- **Multigraph** - Graph (directed or undirected) where multiple edges can connect the same pair of vertices — parallel edges. Multigraphs can represent scenarios where many connections or relationships exist between the same pair of nodes.

There are some basic concepts related to graphs that are important to know:

- **Size** - Total number of edges in the graph.
- **Self-Loop** - An edge that connects a node to itself — it starts and ends in the same node.
- **Path** - Sequence of nodes in a graph where each node in the sequence is connected to the next one by an edge. A simple path is a path where all nodes in the sequence are distinct.
- **Cycle** - It is a path where the sequence's first and last nodes are the same.
- **Subgraph** - A subgraph is a graph generated by extracting a subset of the nodes and edges from a larger graph. In other words, a subgraph is formed by picking a subset of nodes and edges from a larger graph while maintaining the connections and relationships between the chosen nodes.
- **Connected Component** - It is a maximum subgraph (a subgraph that cannot be extended further) where every node is reachable from every other node within the subgraph.

Detailing more the analysis of the connected components, a *connected component* in an undirected graph is a subgraph in which there is a path between any two nodes and no edges connecting nodes in the subgraph to nodes outside. Conversely, a *connected component* in a directed graph is defined as a subgraph in which there is a directed path between any two nodes and no directed edges connect nodes inside the subgraph to nodes outside of it [43].

2.2 Summary

Throughout this chapter, we explore a set of concepts that will prove helpful in understanding the content of this document correctly.

In Section 2.1 (p. 6), we define the graph concept and delve into several specific terms needed to understand our solution.

Chapter 3

State of the Art

3.1	Methodology	8
3.2	Results	9
3.2.1	Conflict Detection	9
3.2.2	Conflict Resolution	15
3.2.3	Conflict Detection and Resolution	16
3.2.4	Surveys and Reviews	24
3.3	Categorization	28
3.4	Analysis	28
3.4.1	Publication History	29
3.4.2	Result Analysis	29
3.5	Conclusions	30
3.6	Summary	31

This chapter analyzes the current systems capable of detecting conflicts, resolving conflicts, or both simultaneously in smart environments.

3.1 Methodology

To better perceive the existing conflict detection and resolution approaches, we used a strategy called snowballing. This technique allows us to use the reference list of a paper or the citations to the paper to find additional papers that may add new contributions [44]. Starting from a review done by Resendes *et al.* [32], where the authors review the existing literature concerning conflict detection and resolution in smart homes, we then expand our search by analyzing all paper citations and all papers that cited this one. To decide whether a result is relevant, we first analyze its title, abstract, year, and keywords. If more is needed, we look at its introduction and conclusion to check if it suits and is relevant to our topic. We have analyzed papers that cite/are cited by this one from Resendes *et al.* [32]. We also add some cited by FortClash [8] and FORTNIoT [7], papers that result from the search expansion.

3.2 Results

Here, we analyze the selected papers, describing the main contributions of each one. We divide this section into four subsections. Section 3.2.1 (p. 9) explores tools that can only detect conflicts. Following that, in Section 3.2.2 (p. 15), we look at some approaches capable of resolving conflicts. In Section 3.2.3 (p. 16), we present solutions that can simultaneously detect and resolve conflicts. Finally, in Section 3.2.4 (p. 24), we explore some surveys and reviews about the topic.

3.2.1 Conflict Detection

In this section, we present five tools capable of detecting conflicts.

Fu et al. [18] aims to improve the detection of rule conflicts in smart homes by dividing the rules into two categories: (i) **explicit conflicts** arise when the same rules' actuators are used, and they are put into action at the same time; (ii) **implicit conflicts** arise when different rule actuators are used and put into action simultaneously. Based on this division, they identified five types of conflicts presented in the Table 3.1 (p. 9).

Category	Types of Rule Conflicts	Conditions for Judging Rule Conflicts
Explicit Conflicts	Opposite Execution Actions Conflict	The execution of rule 1 and rule 2 is opposite
	Conditional Coverage Conflict	The condition of rule 1 contains that of rule 2, or the condition of rule 2 contains that of rule 1
	Conditional Mutex Conflict	The conditions of rule 1 and rule 2 are opposite
Implicit Conflicts	Rule Loop Conflict	The execution action of rule 1 is the same as the condition of rule 2, or the executive action of rule 2 is the same as the condition of rule 1
	Environmental Conflict	The environmental state after executing rule 1 is the same as the condition of rule 2, or the environmental state after executing rule 2 is the same as the condition of rule 1

Table 3.1: Rule conflict types and conditions to judge them

The authors proposed a Conflict Detection Algorithm Model (CDAM) to detect conflicts easily. This algorithm detects conflicts between rules without defining a relation between them. So it can be possible to create a more efficient system, they take time factor into account in the rules. Also, users must classify the rules into one of the available categories (smart security, environmental detection, brightness adjustment, video control, and electrical control) when they create them, considering the device specifications and goals. Then,

users can create priorities between the before-mentioned categories according to their relevance in the smart home (*e.g.*, smart security has a higher priority than brightness). This way, the possible conflicts are between rules of different types with the same priority. To further reduce the intricacy of conflict detection, users should also define the rule execution area when configuring device linkage rules, as well as set user rights, creating priorities between the different users of the smart space in this way. Thus, the conflict detection module will draw the rules in if they have the same priority but are of different kinds.

After formulating the device linkage rules, the system has to extract the rule's critical information. To do that, the algorithm resorts to the Drools¹ rule engine. This information is then added to the rule information verification database, determining if the detection rules meet the same priority, user rights, and execution area. For CDAM to be able to detect conflicts, rules are modeled in format $Rule_{Id} = \{Time, Sensors, Environments, Actuators\}$. Based on conflicts previously defined in Table 3.1 (p. 9), CDAM checks if any conflict is present by comparing the late-created rules with the rules already in the database, as there is no conflict between the last ones.

The authors conducted a comparative study between their solution and UTEA [39], another conflict detection solution. They find that their solution takes less CPU and memory resources, as UTEA needs to judge the relationship between rules. Their approach also decreases the missed and false detection rates compared to the previous method. A possible improvement for this system is making it identify rule conflicts without specifying what kind of conflicts they are.

FORTNIoT [7] presents a solution for poor manageability in home automation technology. This solution produces three main contributions: (i) a well-defined approach that uses self-supporting predictions to simulate the behavior of smart homes; (ii) a visual demonstrator that presents these predictions in a comprehensible way; (iii) a study that determines the impact of the predictions on the user's understanding and trust in the smart home.

FORTNIoT predicts the future behavior of the smart home based on simulations of trigger-condition-action rules complemented by self-sustaining predictions, such as weather forecasts. The authors consider that predictions can be divided into two types, fixed and continuous. Fixed predictions can be predicted only once, as they are unaffected by rule executions or other entities. In contrast, continuous predictions depend on the context and can be affected by other factors, so they must be updated throughout the simulation. The predictability and accuracy of near-future predictions are better than those about a far future. The context also affects the quality of predictions, so new predictions are made when there is a context exchange. The availability of existing services is also considered, and if they do not exist, internal models can approximate an entity's behavior. A vast set of models can be selected, from mathematical formulas to machine learning models.

¹Drools, <https://www.drools.org/>

So that the system can make predictions using trigger-condition-action rules, they need to be adapted and decoupled. Triggers must be able to detect “virtual events”, conditions must evaluate based on entity states, and actions must be saved virtually instead of being executed directly on entities. The approach deduces predictions using three main data structures: queue (a temporally ascending list that records all entity state changes that the algorithm has not yet taken into account), snapshot (a grouping of all entity states at a certain moment), and future data structures (stores all forecasts regarding rules and the state modifications that those rules’ actions will bring about). The algorithm is divided into four phases: initialization, updating self-sustaining predictions, deducing predictions, and tracking causes and effects. It updates self-sustaining predictions and deduces predictions by checking which rules will be triggered, evaluating their conditions, and yielding virtual states. The future predictions queue and snapshot are updated with the new expected states. FORTNIoT also provides users with explanations about what is happening in the smart environment. Figure 3.1 (p. 11) graphically represents this algorithm.

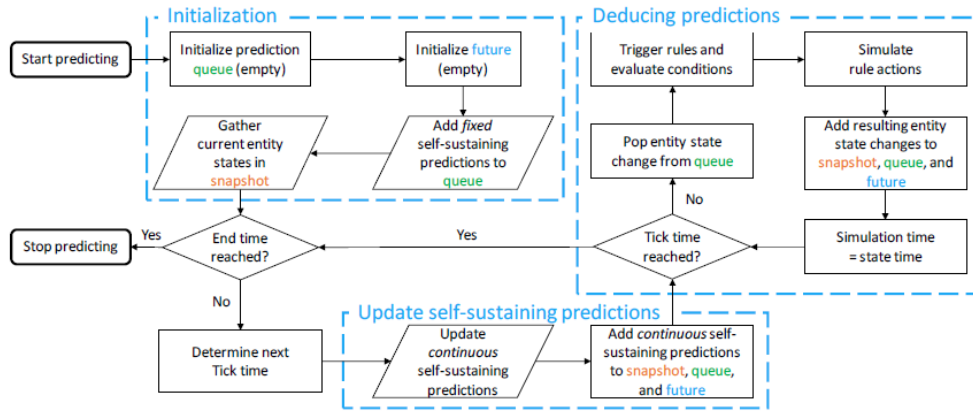


Figure 3.1: FORTNIoT algorithm to deduce predictions

Home Assistant², an open-source middleware that combines IoT entities, was used as the basis for implementing the FORTNIoT algorithm. The architecture has three main components: Rule Manager, Adapter Manager, and Prediction Engine. The Rule Manager manages all rules and explains why conditions are evaluated as true. The Adapter Manager keeps track of the adapters that collect context information for new self-sustaining predictions. The Prediction Engine uses both managers to perform the prediction algorithm.

The authors also design and implement an interactive timeline to increase intelligibility levels and help users understand actions’ effects and why without too much know-how about the system’s inner workings. The two main goals of the visual demonstrator are: (i) to display the predicted future states of entities in the smart home and (ii) to display the reasons why these entities will change.

²Home Assistant, <https://www.home-assistant.io/>

Finally, a study was performed with 42 participants, and it allowed the authors to draw some conclusions:

- As a result of the FORTNIoT prediction system's increased System Causability Scale scores, smart home behavior is better understood. Participants found that the additional information allowed them to answer questions more accurately without making predictions themselves;
- Due to a better understanding, FORTNIoT predictions boost confidence in smart house behavior. Participants in FORTNIoT demonstrated reduced overtrust due to the improved understanding provided by the system;
- FORTNIoT leads to increased confidence in smart home behavior understanding by participants, resulting in higher confidence levels in both open-ended *what* and *why* questions and binary assessments compared to baseline participants.

EUDebug [9] is a solution to help end-users debug their trigger-action rules. It aids users by assisting in identifying rule conflicts and through step-by-step simulations to illustrate the rules' run-time behavior. The authors generalize three categories of issues — loops, inconsistencies, and redundancies — that can originate unintended behaviors and then have to be considered in trigger-action rules. The system is composed of two components: (i) a Semantic Colored Petri Net (SCPN) to model, check, and simulate trigger-action rules, and (ii) a web-based interface for composing rules, showing detected problems, and simulating rules.

A SCPN is a member of the Colored Petri Net (CPN) family, so it combines the power of Ordinary Petri Nets with the advantages of a high-level programming language. This approach helps users in the simulation process and, in contrast with previous strategies, does not consider conditions and analyzes and generates the net using a semantic model. While moving through the net, tokens (used to describe the net's dynamic behavior) adopt different semantic "colors" to help discern between dubious and safe rules. The semantic model employed in this strategy is EUPont [11], offering a high-level ontological representation of trigger-action programming by classifying online services and smart devices according to their categories and capabilities. This ontology details the devices and services that provide triggers and actions and their connections to other triggers and actions. Additionally, triggers and actions are categorized using a class tree depicting their final behavior. Based on their final functionalities, this categorization aids in identifying the similarities and contradictions of triggers and actions. The semantic data is used to express trigger-action rules semantically and to create and examine the SCPN. The model for triggers and actions of an object in the system is represented as places, known as Trigger Place (TP) and Action Place (AP).

The first step to encountering possible problems in rules at composition time is translating them into the corresponding SCPN. Operating a depth-first search on the net allows for finding potential loops. The net must be run and examined using tokens as artifacts to identify inconsistencies and redundancies. One token is initially marked in the TP associated with the established rule, and the activated transitions move that token across the net. An inconsistency is raised when at least two executed actions act on the same device or online service and belong to distinct EUPont classes. If at least two executed actions act on the same device or web application and fit on the same EUPont classes, we are in the presence of a redundancy. Finally, the rules simulation may be carried out step-by-step by randomly choosing a transition to be triggered from the list of allowed transitions.

The EUDebug interface can be divided into Rule Composition, Problem Checking, and Step-by-Step Explanation. The Problem Checking and the Step-by-Step Explanation interfaces enable users to identify rule conflicts and simulate the run-time behavior. The Rule Composition interface is modeled after IFTTT³ to make it user-friendly and accurate in composing rules. To compose a rule, the user selects a trigger and action service, fills in any required information, and analyzes the rule for any problems by the system. If any problem is found in composed rules, users may use Problem Checking and Step-by-Step interfaces to understand it better.

A study with 15 participants was conducted to evaluate the effectiveness of EUDebug in understanding and identifying problems in trigger-action rules. The findings showed that:

- Participants had distinctive perceptions of the different problems, with loops and inconsistencies being considered dangerous while redundancies were mostly accepted;
- Redundancies in online services were perceived as more irritating than those involving physical devices;
- Loops turned out to be the most challenging issue to understand, and the step-by-step explanation was frequently used to comprehend it;
- Indirect issues brought on by implicit activations between rules were equally challenging to comprehend and recognize;
- The highlighting of the detected problem was not sufficient in most cases. Therefore, users were stimulated to comprehend the issues via a step-by-step simulation.

DIAC [26] presents a comprehensive classification of inter-app conflicts in the IoT and introduces new conflict definitions that address critical limitations in existing definitions. It argues that previous definitions oversimplify inter-app conflicts, leading to inadequate conflict detection methods. The paper proposes an enhanced definition reflecting users' experiences and intuitions to address this inadequate detection. The conflicts are categorized into strong, weak, and implicit conflicts based on their distinct characteristics. The paper also introduces the

³IFTTT, <https://ifttt.com/>

IA Graph, a representation for formalizing IoT controls and inter-app interactions. Additionally, an efficient conflict detection algorithm is implemented on a SmartThings (SMT)⁴ compiler, demonstrating significantly improved effectiveness compared to previous solutions. This approach is validated through experiments and a user study.

The IA Graph is a directed graph formally representing IoT controls and inter-app interactions. In the IA Graph, each node represents a state of an IoT app, while each edge represents a transition between two states. These transitions are associated with functions formed by combining a set of implication formulas. These formulas can be transformed into a conjunction norm using first-order logic operations. For example, an implication formula like $switch = 1 \rightarrow light = 1$ is converted to $light \vee \neg switch$ where "light" and "switch" are boolean variables. By utilizing the IA Graph, the paper provides a concise depiction of the controls within each IoT app and the various event schedules.

Conflicts are detected using an efficient conflicts detection algorithm implemented on a SmartThings compiler. The algorithm can detect conflicts of all types using first-order logic and SMT solvers. The algorithm follows a two-step process: first, it constructs the IA Graph, a formal representation of IoT controls and inter-app interplays. Then, it utilizes the IA Graph to detect potential conflicts among independently developed apps intended for an open IoT environment. The algorithm also introduces a method to differentiate weak conflicts, considering variations in device modeling and conflict frequencies, thereby assessing their varying levels of seriousness.

The paper demonstrates that the proposed algorithm outperforms previous solutions by a significant margin through comparative analysis, indicating its enhanced effectiveness in conflict detection.

Huang et al. [20] proposes a generic knowledge graph model to capture the relations between services and environment properties. The authors propose a conflict taxonomy designed for a single-resident smart home setting. Furthermore, they present a conflict detection algorithm that leverages the profiled knowledge graph. Then, they validate the proposed approach's effectiveness and efficiency on real and synthesized datasets.

This novel approach constructs a generic knowledge graph for IoT services using ConceptNet⁵ as the information source. The knowledge graph consists of triplets (h, r, t) , where h represents the head entity, t represents the tail entity, and r signifies the relation between them. Additionally, the paper defines sets of entities E and relations R within the knowledge graph. The constructed knowledge graph represents the relations between IoT services and environment entities.

The study presents a conflict taxonomy to capture various sorts of disputes in the context of a single-resident smart home. The taxonomy contains the following types of conflicts:

⁴SmartThings, <http://www.smarthings.com>

⁵ConceptNet, <https://conceptnet.io/>

- **Function-function conflict:** occurs when two Event-Condition-Action (ECA) rules call a common IoT service feature;
- **Opposite-environment-impact-conflict:** occurs when two IoT services have opposing effects on the common environment variable over the same period and in the same place;
- **Cumulative-environment-impact-conflict:** occurs when two IoT services have the same effects on the common environment variable over the same period and in the same place;
- **Transitive environment-impact-conflict:** occurs when the influence of one IoT service on an environment variable unexpectedly causes the execution of another IoT service.

To detect conflicts, the authors developed the Conflict-detector algorithm. It efficiently discovers conflicts based on the defined taxonomy. The Conflict-detector consists of the “finding overlapping event pairs” module and then verifies which taxonomies fit in the “conflict detection” module.

The authors perform a series of experiments using real and synthesized datasets to validate the proposed conflict detection approach. The experimental results provide evidence of the approach’s effectiveness and efficiency, concluding that this approach can be used to improve the reliability and safety of IoT-based smart homes.

3.2.2 Conflict Resolution

In this section, we analyze a solution for conflict resolution.

Lee et al. [25] address the problem of conflicts in smart homes with multiple users performing different tasks simultaneously. This approach suggests that users use a wearable device with an accelerometer and a gyroscope. This device recognizes the users’ activities through the collected data that are then evaluated for conflict detection and resolution.

The Conflict Resolution Algorithm resolves conflicts between multiple users’ activities in a smart environment. Initially, the authors define default results when there is a single user for variables such as lighting, temperature, and music volume, among others. However, in a multiple-user domain, the default outcomes for these variables may conflict. Thus, two variables are created to enable conflict resolution: “identity” and “time”. The variable “identity” appoints different precedence to each resident based on the assigned value, while “time” customizes the preferred time for a specific activity. The environment’s brightness and temperature are also taken into account. The algorithm first identifies the most conflictual pair of activities for each lighting, temperature, and music volume control setting. Then the maximum tolerance values are adopted for each conflicting activity, which users can easily customize. This procedure can be applied to any service beyond controlling lights,

temperature, and music volume and can be modified based on the specific properties of any IoT service.

The meaningful advantage of this approach is its low cost. Some suggestions for future investigation are (i) detection of more activities and (ii) gathering more information about user experience to establish the maximum tolerance value for each service more accurately.

3.2.3 Conflict Detection and Resolution

In this section, we resume solutions capable of detecting and resolving conflicts.

FortClash [8] approach to unintended behavior in smart homes is to wait until the unintended behavior is expected to occur and then specify a solution by making one-time exceptions. The authors clustered the different types of unintended behaviors found throughout their analysis of the existing literature into nineteen unique types.

FortClash uses a two-step process to predict unintended behavior in smart homes. The first step involves robust predictions of future behavior using a depth-first algorithm that simulates when rules will trigger and deduces the effects of their actions “layer by layer”. These simulations resort to predictions from external services (*e.g.*, weather forecasts) for more reliable results. FortClash algorithm is aware that the order in which rules are inferred can affect whether they are triggered or not. The algorithm runs in a breath-first way to prevent possible race condition problems and guarantee that every unexpected behavior is properly predicted. The prediction is made “layer by layer”, starting from a layer representing each device’s current state. In the second step, “detectors” analyze the predictions and then alert users if any inadvertent behavior is discovered. FortClash has a dashboard where all predictions are presented to the users and helps them better understand what is happening in the smart system. This information, including unintended behavior warnings, is shown in an interactive timeline.

This approach contrasts with previous ones that require all-encompassing solutions without conflicts or need manual corrections right after unexpected behavior occurs. In FortClash, conflict mediation lets users make one-time exceptions for imminent conflicts. There are four different mediation techniques:

- suppression of an action: when the relevant rule is executed, the action will not be carried out, and its effects are removed from the predictions;
- modification of an action: run an action with different values if suppressing its execution is too strict. The repercussion of this adjusted action execution is used in the remaining steps of the prediction process;
- manual planning of an action: if the rule fails to be triggered, users can manually arrange the action and compel it to be executed;
- ignoring potential unintended behavior: user ignores the warning when received.

The authors implemented a feedforward visualization to avoid trial and error in finding a solution. This visualization shows the user how future events will change if an exception is confirmed. The prediction engine has two modes: default and simulation. Both modes collect predictions from external services, use rules to make predictions, and consider the lists of suppressed, modified, and scheduled action executions. Nevertheless, these lists are different depending on the mode. The lists are persistent in default mode, though volatile virtual modifications based on feedforward interaction are also admitted in simulation mode. This peculiarity allows the simulation of the effects of one-time exceptions.

When looking at the predictability of unintended behaviors, the breath-first prediction algorithm provides complete prediction outputs that enable pattern recognition of unintended behaviors. This algorithm is vigorous against loops and inconsistent or redundant actions and ensures that all possible scenarios are predicted, so race conditions are not a problem. Even so, more extensive models are required to detect inconsistent effects or predict when an entity will become unavailable. Unsatisfiable trace conditions, overtrust in the triggers, the rule set overtrust, and redundant events are unpredictable.

Editing trigger-action rules can resolve most of the conflicts. However, these changes will affect future smart home behavior. Nevertheless, some unintended behaviors, like unavailable entities, cannot be avoided by modifying the rules.

CityGuard [27] is a safety-aware watchdog architecture conceived to detect and resolve conflicts between multiple smart services in smart cities. It performs as a feedback loop and is designed to be a middle layer that intercepts actions taken by smart services, detecting and resolving possible conflicts.

The spectrum of conflicts in smart cities can be divided into two groups: (i) device conflicts, when many actions are performed concurrently on a single device, and these actions conflict with one another and, (ii) environmental conflicts, generated directly or indirectly by adverse or conflicting environmental influences. The system consists of four components (*cf.* Figure 3.2, p. 18):

- **City Safety and Performance Requirements (CSPR)**: furnishes the rules to scrutinize all the actions. To decide if actions are safe, CityGuard simulates and evaluates if the effects of an action on the metrics are within their requirements.
- **City State and Service Action (CSSA)**: acquires real city-states and hands them to CityGuard SUMO in the CDR. This way, the real city and the simulated city access the same set of states. In addition, it saves and supplies both incoming and ongoing city actions.
- **Pre-processing**: inspects action's information and handles any missing information. In this phase, actions are also classified, considering their timing: Incoming, Ongoing, and Past actions. The actions are deemed safe single actions following pre-processing and are then forwarded to the CDR.

- Conflict Detection and Resolution (CDR): the system's main component. Includes Simulation of Urban MObility (SUMO), a traffic simulator that simulates the actions' effects on a real city model. Furthermore, it holds the Overall Resolver (OR), where definitive judgments about possible conflicting rules are taken.

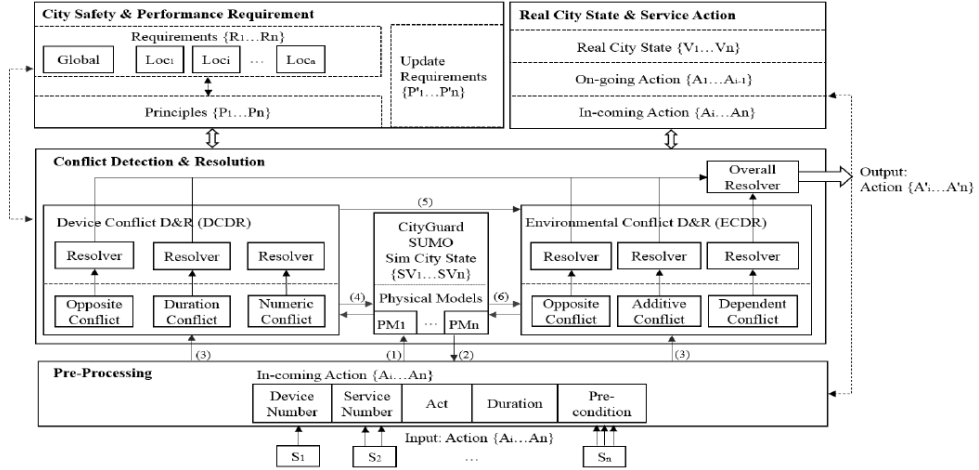


Figure 3.2: CityGuard Structure

To make decisions when resolving conflicts, CityGuard presents two principles: priority-based and performance-based. In priority-based, each domain has a priority based on its relevance to the city. Differently, in performance-based, the performance result of each action (a value that results from the simulation of each action through SUMO) is considered in case their domain is the same. CityGuard aims to guarantee that only safe actions are carried out in the city and can reduce safety infractions and enhance overall city performance compared to standard procedures.

Shah et al. [35] describes a strategy to identify conflicting and incomplete rules and then two strategies for resolving them. In this approach, the authors consider that actions in an IoT system can shift the system's status from one state to another. An "anti-action" is necessary to get back to the initial condition. An "anti-trigger condition" is what causes this anti-action to occur. This system's rule is composed of a trigger-rule and an anti-rule. In this paper, in addition to the usual definition of conflicting rules, they also define incomplete rules. A set of rules is incomplete if a sensor has a range of values for any actuator for which the actuator's action is not stated. The anti-rule provides a way to construct a complete rule.

The first step in this solution is to decompose rules to Disjunctive Normal Form (DNF). DNF is a boolean disjunction of two or more conjunctions (*e.g.*, $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee x_5$).

The rules must be classified in the second step to establish relationships between them. The authors divide rule conflicts into execution conflict, shadow conflict, and independent conflict. Execution conflict occurs when two rules have overlapping sensor value ranges and

conflicting actions. Shadow conflict identifies overlapping rules by examining if the trigger conditions of one rule are contained within the trigger conditions of another rule and if both rules have the same action. Independent conflict occurs when two rules have different sensors and conflicting actions. After being converted to a DNF, the rule comprises several clauses, and the relationships between these clauses determine if the two rules conflict. The authors also suggest a way to detect rule incompleteness. A complete rule must have entire clauses in a smart home system rule. At least one sensor must be able to cover the whole range for a clause to be considered complete. The covering polygon problem, a well-known computational geometry problem, is a variant of the challenge of determining if such subsets of clauses exist that complete a sentence. When determining if a set of rectangles completely encloses a particular polygon, the system does so. That is accomplished by gradually expanding the collection of non-overlapping clauses until all potential sensor ranges are covered. The clause is considered incomplete if, after adding all possible clauses, the range of at least one sensor remains uncovered.

The article covers two approaches to addressing rule conflicts in a smart home system. The first approach is the priority method, wherein each rule is given a priority, and in the case of a conflict, the rule with the highest priority is carried out. The second technique involves changing the rule sensor values, which addresses the conflict by changing one or more sensor values. According to the paper, altering rule sensor values is appropriate for resolving execution conflicts, whereas the priority technique is appropriate for addressing independent conflicts. Simply eliminating the rule that is a subset of the super rule resolves shadow conflicts.

Ahmed *et al.* [2] introduce a specific plan, called the Agent-Based Negotiation Approach (ABNA), for identifying and addressing Feature Interactions (FIs) (unintended behaviors) in Smart Home Systems (SHSs). The ABNA approach models the elements of SHSs (services and inhabitants) as agents to establish, if possible, a compromise between conflicting objectives, a struggle for shared resources, and missing connections between services.

The ABNA mechanism for handling conflicts in SHSs operates through a negotiation system. The ABNA mechanism uses a negotiation system for addressing conflicts in SHSs. The objective of ABNA, in contrast to previous methods, is to establish a compromise by enabling competing services to operate concurrently with somewhat lower efficiency without harming overall performance. This is preferable to a total service disablement or favoritism of one person over another. ABNA uses agents to enable the simultaneous operation of services with conflicting features. Compromise acceptance is based on agent-defined features and preferences, ordered hierarchically. A negotiation session between the involved agents is set up by ABNA whenever a task has to be performed. The initiating agent, negotiator agent, and participating agents are present during the session. The engaged agents assess the proposal and reply by stating whether it can be accepted, allow a compromise, be counterproposed, or be rejected. Each proposal has three parts: the device ID, the action

that will be taken, and the value used to define the action's degree. A counterproposal is required and will be the agent's lowest priority job if it is impossible to accept the initial proposal or enable a compromise on it. In Figure 3.3 (p. 20), it is possible to see this algorithm graphically represented. ABNA's specification is formalized through the Calculus of Context-Aware Ambient (CCA), which enables the examination of ABNA through its execution environment.

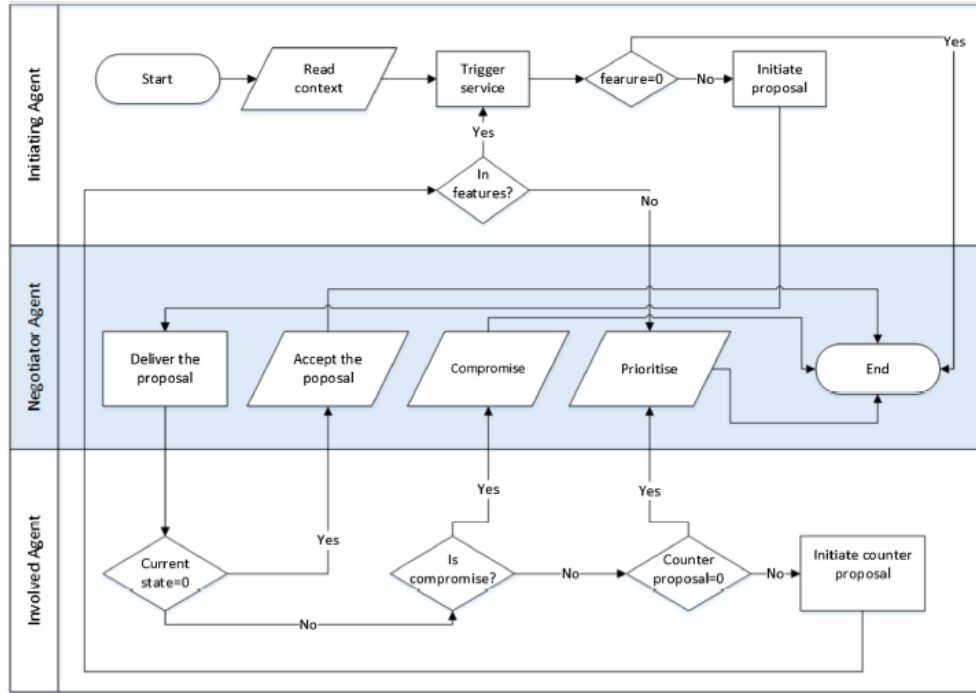


Figure 3.3: Negotiation algorithm proposed by Ahmed *et al.*

The authors present a case study to put into practice the concepts outlined previously, showcasing the practicality of ABNA, and then examine the accuracy of ABNA using the ccaPL (execution environment of CCA) formalism.

AutoTap [45] is a new system that makes it simpler for customers to configure online services and smart devices. By letting users set both rules and properties that the system should always satisfy using graphical interfaces, it almost removes the possibility of human error. Users may define the properties they need the system to have rather than create event-driven rules. AutoTap will create rules from scratch that fulfill properties if no rules are given. If initial rules are defined, AutoTap will review them and make any required modifications to ensure the system does not violate the property. An innovative property-specification interface and novel TAP rule modeling methods from pre-defined properties are AutoTap's main building blocks.

The techniques employed in AutoTap are novel and general and cannot be reproduced by the previously existing synthesizers. Figure 3.4 (p. 21) shows an overview of this solution. To accomplish its goals, AutoTap takes three steps:

- **Model Construction:** Transform the existing smart device system, its TAP rules (if any exist), and the desired property δ into a Büchi Automaton A that accepts executions that violate δ , similar to how traditional model checkers process internally;
- **Patching the Automaton:** Determine how to alter the Büchi Automaton A so that all executions that meet the desired property δ are maintained, ensuring compatibility, and all previously accepted (i.e. violating δ) executions are eliminated, ensuring compliance with the property;
- **TAP Synthesis:** Determine the valid TAPs that can implement the changes to the automaton proposed in the previous step.

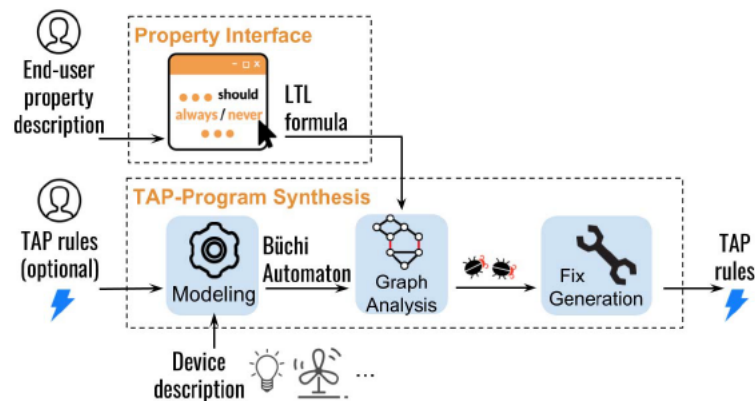


Figure 3.4: AutoTap overview

The authors also carried out two user studies. The first one intends to identify the most often demanded features for smart devices. The second one pretends to assess the ease of use of AutoTap’s approach of expressing intent through properties and to examine the ability of novice users to correctly communicate their desired outcomes compared to the traditional TAP rule interface. From the first one, they discovered that seven pre-defined templates encompassed the majority of the properties that the participants wanted for their smart devices. The second one showed that users preferred using AutoTap over a traditional TAP rule interface, mainly when several trigger-condition-action rules are required to cover every conceivable case.

AutoTap has limitations when it comes to issues such as actions not finishing or not taking effect right away, which call for a more detailed model from device manufacturers. Users are still susceptible to making errors while writing properties, which might result in undesirable outcomes or conflicting attributes that AutoTap cannot resolve.

Braga Silva et al. [37] propose a solution for conflict detection and resolution in ubiquitous and context-aware systems with multiple users. Generally, users in the same environment have common goals. However, challenges in these collective contexts emerge due to differences in individual profiles and/or limited resources. In this work, a centralized architecture to

handle these conflicts that takes into account individual and collective efficiency is presented. This solution is also a flexible and robust service with low resource consumption.

The authors propose an enhanced client-server architecture model more appropriate for conflict detection and resolution in ubiquitous devices. This approach defines a server rotation scheme where all group members take up the server's responsibilities for a specific time, ensuring each device uses resources similarly. As a result, the problem of the rapidly approaching depletion of the server's energy source is resolved. The suggested architecture works within the communication network established among the devices. The Conflict Engine framework (*cf.* Figure 3.5) has been defined to detect and solve collective conflicts with the help of the modules that make it up and are responsible for gathering context information. The Conflict Detection Module detects impasses among participants during task adaptation by conducting three-dimensional research that considers users' profiles, contexts from shared environments, and the application's tasks. The Conciliation Module is informed of possible conflicts detected in the previous step and adjusts the tasks according to individual and collective interests while considering resource availability. This module is responsible for conflict resolution by executing a resolution algorithm or technique considering collective and context characteristics. Finally, the Re-startup Module is responsible for data exchange among the devices. This framework is generic and does not specify the implementation of each module.

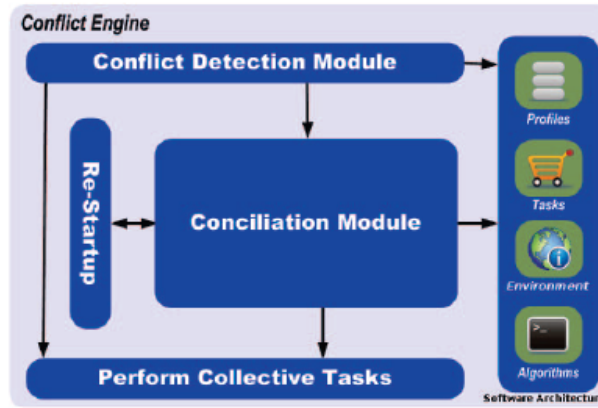


Figure 3.5: The Conflict Engine framework

Furthermore, this work proposes a novel methodology to solve collective conflicts in ubiquitous and context-aware applications. The methodology consists of Conflict Engine modules with defined behaviors that are flexible and parameterizable for different applications. Each application must provide its Action Level, which contains information on dimensions to consider for conflict detection, resolution algorithms, and their parameters. The Conflict Detection Module analyzes the input dimensions furnished by each application to check the circumstances that lead the system to an inconsistent state. Next, the module analyzes that combination to detect whether conflicts are present. In case of conflicts, the Conciliation

Module is activated, which selects the best conciliation algorithm from its repository, considering the application's QoS and resource consumption demands. The selected algorithm may need to be adjusted, and its execution is supervised to ensure expected QoS and energy consumption levels. A penalty is applied to influence future algorithm selection if an agreement is violated, affecting future selection.

This solution is evaluated by simulating its application in different contexts with different needs. The results showed that this dynamic approach successfully improved the balance between user satisfaction and resource management. In future work, the authors propose implementing new algorithms for the methodology and further evaluations with other applications.

Bu et al. [5] researches new methods and resources to help non-expert IoT users systematically construct high-confidence real-time HA-IoT systems. Two central tenets underlie this system: (i) increasing number of IoT devices operating in continuous real-time contexts with dynamic rules and time delays and, (ii) the user should be as engaged as possible during the system modeling, specification verification, and violation fixing processes. Contributions of this system include: (i) Hybrid Automata Model Checking of Real-Time HA-IoT System to model and check the system behavior and help in debugging (ii) Counterexample-guided Fix Suggestions (iii) System Implementation and Real Case Evaluation through MenShen IoT programming platform

This framework aims to help users automate their homes and boost their confidence in the system. It can be divided into three parts: Linear Hybrid Automata (LHA) Automatic Modeling, Reachability Analysis of LHA Model, and Counterexample-guided Fix Suggestion Synthesis (*cf.* Figure 3.6, p. 24). In the first step, the LHA models are generated based on device documentation and IFTTT rules, as asking users to build them is unfeasible. Next, in step 2 occurs the check of the user-specified undesirable reachability specification (previously defined and in addition to rules) using the path-oriented Bounded Model Checking (BMC) technique. Also, this step returns an error trace in case an unwanted target is reachable. Finally, the last phase helps debug the counterexample in case of model failure. The framework supplies guidance (*e.g.*, fix suggestions) and presents a method that first parameterizes the system and then solves the free parameters using Quantifier Elimination strategies. A counterexample-guided strategy is defined based on finding a value range that dismisses the founded counterexample, selecting a value on this range as a fixed suggestion.

MenShen implements this framework suggested by the authors and supports the earlier-mentioned functionalities. MenShen can check and fix real cases with a vast set of rules and devices, representing a massive improvement in time spent on this analysis. Users can interact with MenShen to select specific rules to fix or give favored ranges of rule parameters.

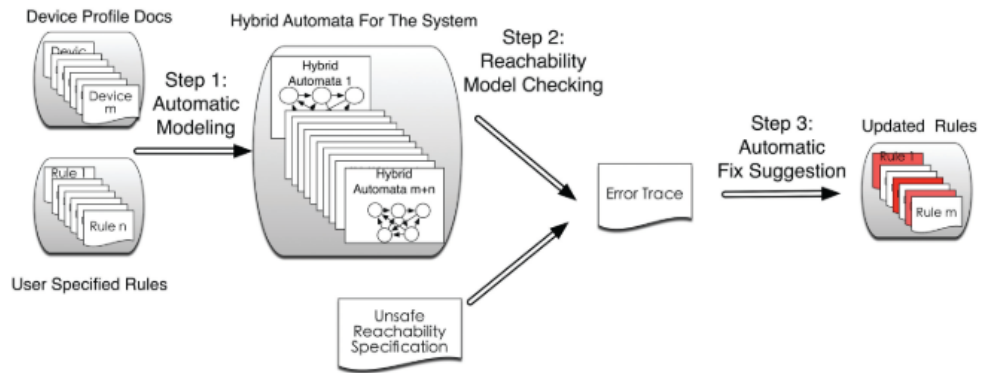


Figure 3.6: Framework Architecture

3.2.4 Surveys and Reviews

In this section, we summarize some surveys and reviews that we found relevant while searching.

Resendes *et al.* [32] make a systematization and review of the existing literature about conflict detection and resolution in multi-user Home Building Automation Systems (HBAS). Their motivation for working on this topic is that it still needs to be explored and has much potential. Furthermore, the existing literature on it is considerably sparse and disorganized.

The authors begin by making some considerations about HBAS. One of the most critical points is that it is of absolute importance that a HBAS does not interfere nor interpose in the user's routines. To make it possible, the smart environment needs to be context-aware. Sensors must collect and share data about what surrounds them to make informed decisions. Next, they identify and describe the components that compose a smart home (*e.g.*, actuators, sensors, controller) and then share an example of a conflict and its consequences. Figure 3.7 (p. 25) is an overview of an intelligent environment containing sensors, actuators, and a controller that makes decisions based on gathered data.

After this introduction, the authors dedicate a chapter reviewing some basic concepts of HBAS and explaining how data flows and how this data is used. Meanwhile, they suggest possible uses of HBAS, praising the importance of making users feel they control the system and that users must be the central piece of these systems.

The following chapter focuses on Ambient Intelligence (AmI), defined as the “paradigm of creating environments that are able to, as silently and imperceptibly as possible, detect and respond to users' needs, without requiring any explicit orders from them”. The environment resulting from the interconnection of these devices has some important characteristics: (i) embedded in the environment; (ii) context-aware; (iii) personalizable; (iv) adaptive to the user and environment changes; (v) predictive. The rest of the chapter examines the context-aware definition, associated challenges, and Human-computer interaction.

The fourth chapter targets conflicts. It starts by presenting some *conflict* definitions and then classifies them, taking into account the following dimensions:

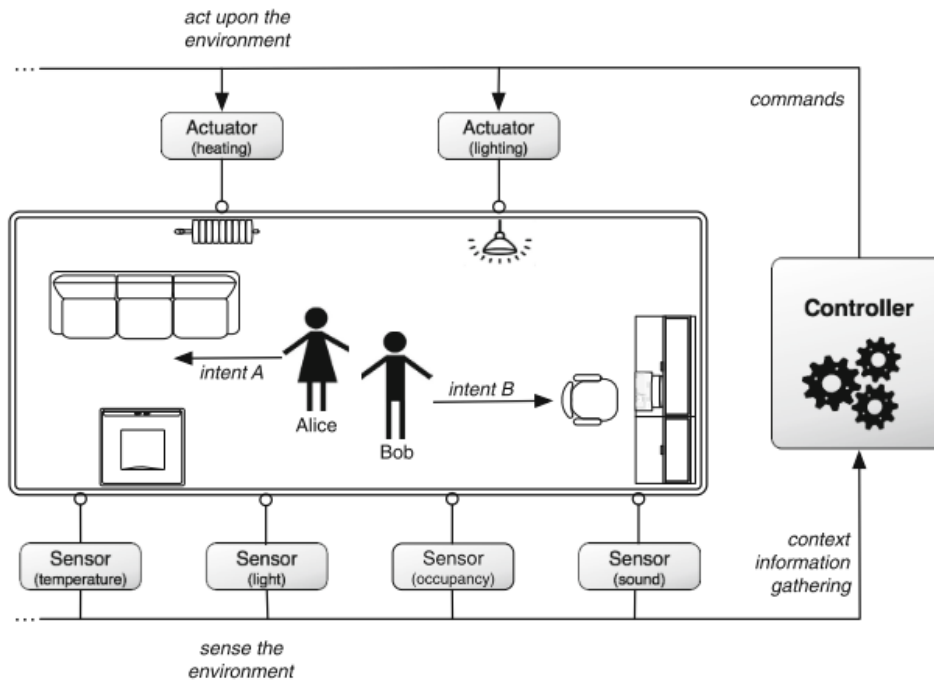


Figure 3.7: Intelligent environment overview

- **Source:** Resource, application, policy or role/profile;
- **Intervenients:** Single user, user vs. user or user vs. space;
- **Time of detection:** *a priori*, when it occurs or *a posteriori*;
- **Solvability:** Conflict avoidance, conflict resolution, acknowledge inability or acknowledge occurrence.

Still, in the same chapter, the authors address the origin of the interpersonal conflicts (values, relationships, external/moods, data or structure) and then explore and adapt an interpersonal conflict handling proposed by Thomas [41] (*cf.* Figure 3.8, p. 26). This model can be applied to device conflicts. At the end of the chapter, the authors present and analyze some conflict resolution approaches for AmI.

Finally, some conclusions are drawn to finish the work, and some gaps are presented. The authors conclude that a system capable of automatically resolving conflicts has to be aware of the surrounding context, so it can check for conflicts and make decisions based on the users' requirements. One of the significant gaps in this area is the need for a system that can immediately and correctly react and decide in decision-demanding situations.

Pradeep *et al.* [31] provide a comprehensive survey of the existing literature on detecting and resolving conflict in IoT systems. The authors outline the underlying issues and complexities in identifying and resolving conflicts in IoT systems and lay out the future challenges and research directions in this area.

		Cooperativeness		
		low	medium	high
Assertiveness	high	Competition		Cooperation
	medium		Compromise	
	low	Avoidance		Accomodation

Figure 3.8: Conflict handling proposed by Resendes *et al.* that considers two dimensions — assertiveness and cooperativeness. Under the diagonal are represented destructive resolutions, while above are represented constructive resolutions

The challenges in conflict detection highlighted in this paper include the lack of visibility of IoT systems (challenging to obtain a comprehensive view of the system), complexity, dynamic behavior, scalability, and heterogeneity (devices and applications use different communication protocols, data formats, and security measures).

The paper emphasizes several challenges associated with conflict resolution in IoT systems, including complexity, trade-offs, real-time constraints, lack of standards, and privacy and security.

Figure 3.9 (p. 27) provides an overview of the type of conflicts and the conflict detection and resolution methods discussed in this paper.

The authors divide conflicts into three types:

- **Rule-Based Conflicts:** arise when the automation rules or policies desire to conduct opposing or otherwise unwanted actions on the same type of device. These conflicts

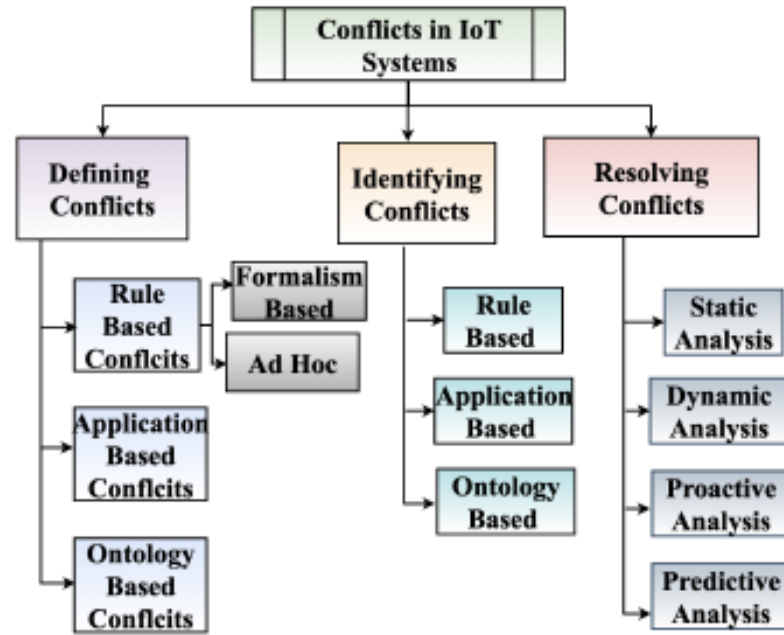


Figure 3.9: Overview of conflict types and conflict detection and resolution methods proposed by Pradeep *et al.*

be further classified in *Formalism-Based* (define a set of safety properties or a policy) or *Ad Hoc* (flagging a particular behavior);

- **Application-Based Conflicts:** arise when many IoT apps attempt to access the same device or have different/opposing impacts on the same device or environment.
- **Ontology-Based Conflicts:** arise when IoT services are defined using ontologies and different kinds of conflicts are detected based on these IoT services' functional and non-functional properties.

The authors divide these systems into the same categories as done for conflict types to identify conflicts. For each of these categories, they present some tools and existing mechanisms to identify each conflict type.

For conflict resolution, the authors classify the existing methods into four types:

- **Static Analysis and Resolution:** static analysis tools generate a semantic model of the program at compile time without running it and then examine different features of the model. Static conflict resolution is made offline, and it is important to identify all conflict types that must be detected;
- **Dynamic Analysis and Resolution:** conflicts are checked dynamically for each action, and if a conflict scenario is discovered, it is handled by taking some action, frequently just stopping one of the conflicting actions.
- **Proactive Analysis and Resolution:** a proactive system is a system that anticipates occurrences based on existing conditions for a given period. These conflicts can be

proactively resolved, if feasible, by altering the entities involved. Based on the context, physics of the situation, and maybe the history of prior occurrences, the proactive model must identify events that may occur shortly.

- **Predictive/Analysis and Resolution:** a predictive system is a system that monitors and learns from the system's behavior over time and maybe across several occurrences of a comparable series of events and actions.

Most existing conflict resolution systems are based on static or dynamic analysis and resolution. No works employ prediction (as explained previously) to resolve conflicts. Nonetheless, several works attempt to predict user behavior in an IoT system. Proactive approaches are also scarce.

Finally, the paper highlights several future challenges related to conflict detection and resolution in IoT systems, including balancing visibility and conflict detection/resolution, anomaly detection, access control models, sensor/actuator state verification, and proactive conflict resolution.

3.3 Categorization

The previously described systems presented in Section 3.2 (p. 9) were categorized taking into account the following categories:

1. **Conflict Detection.** Describes how the conflict detection process unfolds.
2. **Conflict Resolution.** Describes how the conflict resolution process unfolds.
3. **Debugging.** Identifies if the system is capable of helping the users understand *why* or *why not* some rule is triggered and in which context.
4. **Multi-User.** Related to whether the solution can manage conflicts in multi-user environments.

Table 3.2 (p. 29) presents the respective categorization of each paper analyzed in Section 3.2 (p. 9).

A brief table analysis allows us to draw some conclusions. Almost all analyzed solutions to resolve conflicts are based on defining priorities and other variables or in one-time exceptions.

Section 3.4.2 (p. 29) presents further analysis of these results.

3.4 Analysis

In this section, we analyze, based on results presented in Section 3.2 (p. 9), how the number of publications about the topic has evolved through the years (*cf.* Section 3.4.1, p. 29). After that, in Section 3.4.2 (p. 29), we draw some conclusions based on Table 3.2 (p. 29).

Work	Year	Conflict Detection	Conflict Resolution	Debugging	Multi-User
Fu <i>et al.</i> [18]	2020	Priorities and rule categorization	-	-	✓
FORTNIoT [7]	2020	Self-sustained predictions	-	✓	-
EUDebug [9]	2019	SPCN	-	✓	-
DIAC [26]	2019	Directed graph for inter-app interactions	-	-	-
Huang <i>et al.</i> [20]	2021	Knowledge graph model	-	-	-
Lee <i>et al.</i> [25]	2019	-	Priorities and environment variables	-	✓
FortClash [8]	2022	Simulation and analyze predictions	One-time exceptions	✓	-
CityGuard [27]	2017	Simulation and analyze predictions	Priorities and simulated performance	-	-
Shah <i>et al.</i> [35]	2019	Decompose rules to DNF and classify rules	Priorities and change rule sensor value	-	-
Ahmed <i>et al.</i> [2]	2019	Negotiation between agents	-	-	✓
AutoTap [45]	2019	Define properties and Büchi Automaton	-	-	-
Braga Silva <i>et al.</i> [37]	2010	Conflict Engine framework	-	-	✓
Bu <i>et al.</i> [5]	2018	Hybrid Automata Model Checking	Counterexample-guided fix suggestions	✓	-

Table 3.2: Results Categorization

3.4.1 Publication History

Looking now at the previously analyzed publications’ (*cf.* Section 3.2, p. 9) distribution through the years, we graphically represent it in Figure 3.10 (p. 30), so a better analysis can be done.

Most of the analyzed articles from this plot are recent (from 2017 to date). The only exception is a paper from 2014, which is the survey from where we started our search, and another from 2010, which presents a conflict detection and resolution system.

3.4.2 Result Analysis

Based on the categorization done in Section 3.3 (p. 28), the following conclusions can be drawn:

Conflict Detection In terms of conflict detection, the presented systems differ greatly, ranging from defining priorities and rule categorization [18] to simulating the environment’s future behavior based on defined rules [8, 27]. Recently, approaches that use graphs have begun to be explored [20, 26].

Conflict Resolution Half of the analyzed solutions are based on the user’s pre-defined priorities [25, 27, 35]. In the other half, one resorts to one-time exceptions [8] and the others to negotiations between the agents [2] or by defining properties that should be satisfied [45].

Debugging Only four solutions help users in debugging process [5, 7, 8, 9], helping to understand how the environment is working.

Multi-User There are few solutions [2, 18, 25, 37] that are prepared to detect and resolve conflicts in multi-user environments.

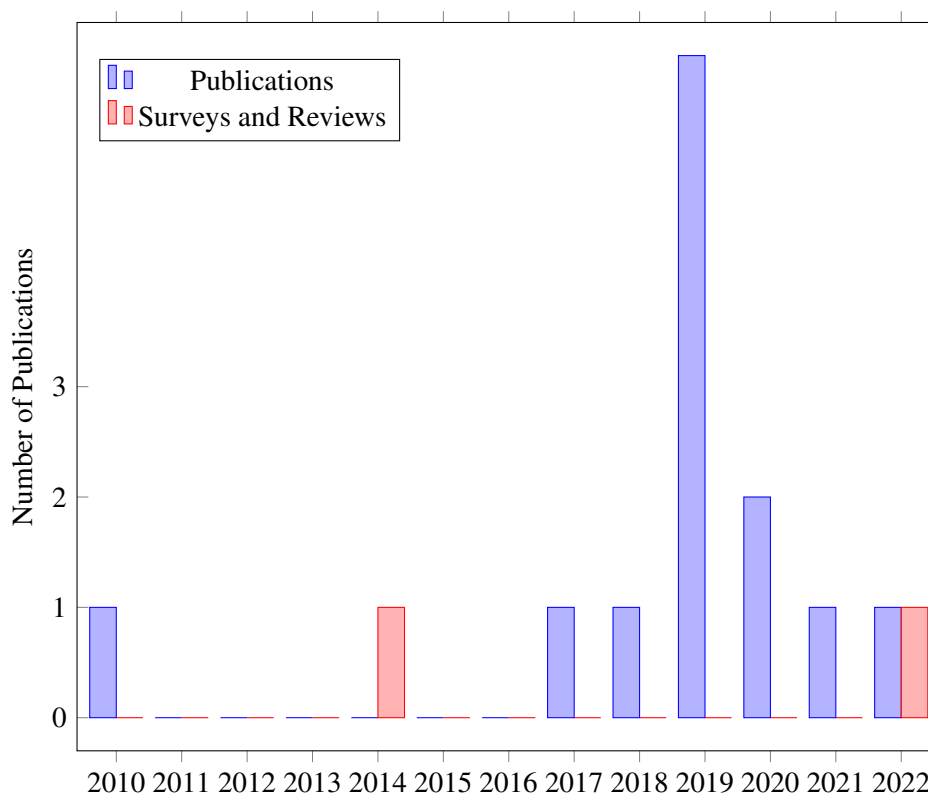


Figure 3.10: Publications per year

3.5 Conclusions

After reviewing some articles that focus on conflict detection and resolution, we were able to have a more in-depth view of the techniques that exist to deal with conflict situations in smart environments. The various existing solutions for conflict detection include defining priorities and categorizing the rules [18], simulations and subsequent analysis to detect conflicting situations [8, 27] or the construction of automata [45]. Despite different solutions for conflicting detection, the area of conflict detection using graphs to represent the IoT environment still needs to be explored more. Although there are already some approaches [20, 26], this path could prove viable as long as the systems are efficient so that the number of nodes does not increase exponentially. Detecting conflicts such as loops, inconsistencies, or redundancies through graphs will be almost instantaneous, as graphs already contain this information.

As mentioned by Pradeep *et al.* [31], there are two basic conflict resolution strategies: (i) *avoidance*, using static analysis to identify conflicts and resolve them by modifying the operational policy logic of the system, (ii) *resolution*, identifying conflicts during execution and implementing measures to resolve them (viz. self-healing [12, 13, 15, 16]). However, this run-time detection when conflicts occur instead of trying to predict them can cause problems. In this way, it is desirable to use proactive or predictive solutions. As mentioned in this paper, these types of solutions are scarce and should, therefore, be further explored so that users are previously aware

of what will happen and can make the necessary changes. As we point out in Section 3.4.2 (p. 29), the bulk of solutions we found for conflict resolution use pre-defined priorities [25, 27, 35] or wait until the conflict is likely to occur to warn the user about it and then resort to one-time exceptions to resolve it [8].

Another relevant core point for our project that is interesting to reveal is that a few solutions are prepared to deal with conflicts in multi-user situations. As Resendes *et al.* [32] stated in their review, there is a need for research in automatic conflict detection and resolution tools for multi-user smart spaces. The greater the number of users, the greater the difficulties in resolving conflicts in a way that satisfies all [32]. Thus, it is crucial to make the users understand the environment and explain why the rules they have defined are not working as expected. This boosts users' confidence in the system and makes them feel they control the smart environment and it works as expected, which is more complicated if unintended behaviors occur. As we noted while studying the systems with debugging capabilities [7, 8, 9], the user better understands the current problems and can easily define new rules.

If we look at privacy questions, they are not even a concern in all presented papers. Data security and privacy are one of the challenges identified by Pradeep *et al.* [31]. As they refer, in IoT systems, conflict resolution may involve sensitive data, such as personal health information or financial transactions. Conflict resolution processes must not jeopardize the privacy or security of this data. Our revision further shows that people look at different types of conflicts differently. They are more worried about loops and inconsistencies than redundancies, and the most challenging task is understanding and solving loops [9]. The current state of the art suggests that there are still many challenges concerning that topic. The existence of many gaps to be explored makes us confident that we can build an innovative solution to ease the rules creation process in smart environments.

3.6 Summary

Throughout this chapter, we carried out a bibliographic review to determine what types of conflict detection/resolution systems currently exist and their advantages or disadvantages. To do so, we performed a search by expansion following the methodology explained in (*cf.* Section 3.1, p. 8). The results obtained in this research are then grouped and explored in (*cf.* Section 3.2, p. 9). Each result is analyzed, verifying the methodology used, how they were implemented, and the improvements and limitations each presents. Results are categorized according to a set of parameters defined by us (*cf.* Section 3.3, p. 28), and a global analysis of the results is carried out, describing the main findings for each of the categories in which the papers were previously grouped (*cf.* Section 3.4, p. 28). Lastly, in Section 3.5 (p. 30), we summarize the main findings from the past analysis, detecting some issues that may be explored in future works.

Chapter 4

Problem Statement

4.1	Current Issues	32
4.2	Desiderata	33
4.3	Hypothesis	35
4.4	Research Questions	35
4.5	Methodology	37
4.6	Assumptions	38
4.7	Summary	38

This chapter describes the problem tackled by this work, the hypothesis it aims to validate, and the methodology we will use. In Section 4.1 (p. 32), we describe the current limitations derived from the analysis we have done in Chapter 3 (p. 8). In Section 4.2 (p. 33), we present, based on the identified issues, our desiderata which are the most crucial points of our work. Section 4.3 (p. 35) presents the hypothesis of our work. Research questions are declared in Section 4.4 (p. 35), pursued by the methodology we will follow in Section 4.5 (p. 37) and an assumption that persists throughout the next chapters (*cf.* Section 4.6, p. 38). Finally, we summarize what this chapter discusses in Section 4.7 (p. 38).

4.1 Current Issues

The sales and adoption of IoT devices by users with distinct programming knowledge have grown massively in the last few years. IoT equipment has many utilities, smart homes being one of them. Here, smart devices are connected and share data between them to monitor the environment and take actions taking into account the context. The main objectives are to provide comfort and well-being and reduce the number of unpredictable behaviors due to rule conflicts as much as possible. These goals raise many different challenges for developers, one of them the conflict detection and resolution.

Throughout Chapter 3 (p. 8), we have analyzed several solutions that attempt to provide better conflict prediction and resolution systems. As we previously stated while drawing some conclusions in Section 3.5 (p. 30), it is possible to identify some issues in current approaches:

1. To our knowledge, no solution can efficiently generate and analyze all possible scenarios from the rules and devices defined in a smart environment. The majority of the current solutions based their approach on debugging past system behaviors to predict possible future behaviors or simulations. Solutions that use graphs to address the problem of conflict detection are rare.
2. Solutions able to detect and/or resolve conflicts in multi-user environments are scarce, and most resort to setting priorities between users. That implies that it is more complicated for a user with a lower priority to define his rules that may interfere with those of a higher priority unless the last ones decide to change their rules. Overall, priority setting can interfere with users' comfort with lower benefits.
3. Most of the analyzed solutions do not inform end-users about detected conflicts and *why* they are conflicts. This explanation is a feature that must be present in this type of system. With today's increasingly complex smart environments, it becomes essential to provide users with information about why a particular conflict occurs or, at least, under what circumstances it will occur.

The identified issues serve as a starting point for us to build a system capable of covering at least part of these problems and build a system that facilitates users' configuration and use of smart spaces.

4.2 Desiderata

Based on the previously identified issues, we have defined a set of goals we want to address in this work:

D1: Identify loop conflicts between rules. A loop conflict occurs when a cycle or circular dependency exists among the rules. In other words, the execution of the rules keeps triggering each other indefinitely, leading to an endless loop.

In this work, we want to detect loops between three or more rules, always bearing in mind the respective periodicity of each one.

D2: Identify inconsistency conflicts between rules. Inconsistencies occur when two rules are triggered (almost) simultaneously, having conflicting actions in the same device. There are two major types of inconsistencies: (i) between rules with the same triggers but different actions in the same device; (ii) between rules with different triggers and actions but actuate in the same device. In this work, we will focus on the first of these types.

When comparing two rules to check if there is any inconsistency between them, we have to consider some factors:

- **rules' triggers and sensors used:** rules sharing the same sensors; rules sharing only part of the sensors; one of the rules has no trigger part; both rules have no trigger part;
- **sensors' type:** *boolean vs. boolean; numeric vs. numeric;*
- **comparison symbol between the sensor and attributed value in trigger:** $=, \neq, <, >, \leq, \geq$
- **comparison between values attributed to each sensor:** $value1 = value2; value1 < value2; value2 < value1; value1 \neq value2;$
- **action of each rule:** rules have the same action in the same actuator; rules have different actions in the same actuator;
- **actuator's type:** *boolean; numeric*
- **periodicity of each rule.**

We will have to analyze the combination of all these factors to see in which cases the rules will be inconsistent.

D3: Identify conflicts caused by redundant rules. Two rules are redundant when both define the same triggers and simultaneously trigger the same action. Redundant rules do not provide additional value or functionality to the system and can lead to unnecessary processing and inefficiencies.

When comparing two rules to check if there is any inconsistency between them, we have to consider some factors:

- **rules' triggers and sensors used:** rules sharing the same sensors; rules sharing only part of the sensors; one of the rules has no trigger part; both rules have no trigger part;
- **sensors' type:** *boolean vs. boolean; numeric vs. numeric;*
- **comparison symbol between the sensor and attributed value in trigger:** $=, \neq, <, >, \leq, \geq$
- **comparison between values attributed to each sensor:** $value1 = value2; value1 < value2; value2 < value1; value1 \neq value2;$
- **action of each rule:** rules have the same action in the same actuator; rules have different actions in the same actuator;
- **actuator's type:** *boolean; numeric*
- **periodicity of each rule.**

We will have to analyze the combination of all these factors to see in which cases the rules will be redundant.

D4: Provide explanations about conflicts. Providing users with an example of a scenario where a set of rules will conflict will make it easier for users to understand the conflict.

D5: Detect conflicts in a multi-user smart environment taking into account privacy concerns.

Normally, in a smart environment like a smart home, several people cohabit and can create new rules without knowing the rules created by other users. Suppose we can detect conflicts between rules created by different users without revealing which user created the conflicting rule or what the rule's specifications are. In that case, we will have a system that considers users' privacy.

D6: Suggest possible ways to solve the conflicts. This way, the conflict resolution task will be much easier, as the system knows all possible scenarios for the smart environment and, based on that, can suggest corrections for the rules that do not create new conflicts.

4.3 Hypothesis

The hypothesis that guides this dissertation is:

“Can a graph-based approach, utilizing nodes to represent all possible scenarios of an Internet of Things (IoT) system, and edges to illustrate the transitions between scenarios driven by the rules, effectively and efficiently detect inconsistencies, loops, and redundancies within the system?”

More specifically, we want to build a system capable of detecting conflicts such as inconsistencies, redundancies, and loops, starting from a set of rules and devices defined for a smart environment. To do so, we intend to use a graph-based approach, representing each node as a possible scenario for existing devices' states used in that environment and each edge as a defined rule, representing the transitions between different scenarios. The same rule may be associated with several edges. To make the system more efficient, strategies to reduce the number of possible scenarios — and, consequently, the number of nodes and edges in the graph — will have to be studied.

It is also crucial to explain some concepts presented in the hypothesis:

Scenario: set of states in which all smart devices in a given smart space are found at a given time;

State: at any moment, according to the respective specifications, each device is in a certain state.

4.4 Research Questions

We present a set of research questions to guide our research towards fulfilling our *desiderata* and help validate or refute the hypothesis.

RQ1. *Is it possible to build a system for a smart home context capable of **generating every possible scenario** containing the states in which intelligent devices can be found in that environment?*

Knowing all the possible scenarios in which smart devices can be found, it will be more straightforward to identify if a set of rules may conflict and in which circumstances the conflict will occur.

RQ2. *Is it possible to build a system for a smart home context capable of **reducing the number of possible scenarios** to make it more efficient?*

The graph size might be an issue when dealing with large-scale IoT systems. The graph can grow exponentially as the number of possible states and rules grows, resulting in an enormous rise in nodes and edges.

Reducing the possible scenarios for a given smart environment will help mitigate the difficulties of processing large graphs.

RQ3. *Is it possible to build a system for a smart home context capable of **predicting loop conflicts** between three or more rules?*

We consider that a loop conflict occurs when some rules repeatedly trigger each other indefinitely in a set of rules, leading to an endless loop.

The following example demonstrates this problem:

- Rule 1: If motion is detected in the living room and living room lights are off, set the living room lights brightness to 50%
- Rule 2: If motion is detected in the living room, set the living room lights brightness to 25%.
- Rule 3: If motion is detected in the living room, turn off the living room lights.

We are dealing with a loop conflict here. Rule 3 triggers Rule 1. Rule 2 is also triggered, selecting a different brightness for the living room lights. With lights on, Rule 3 can also be triggered again, generating a cycle between these rules because the state in which the lights in the living room are will be constantly changing.

RQ4. *Is it possible to build a system for a smart home context capable of **predicting conflicts** between rules with (partially) the same triggers but different actions in the same device?*

We consider that a conflict between rules with (partially) the same triggers but different actions in the same device occurs when two rules (sharing at least one sensor in triggers or having no trigger) are triggered (almost) simultaneously and try to change the state of the same device in a conflicting way.

We can illustrate this with the following example:

- Rule 1: If motion is detected in the bedroom, turn off the living room lights.
- Rule 2: If motion is detected in the bedroom, turn on the living room lights.

In case of motion is detected in the living room, we have a conflict between rules with the same triggers but different actions in the same device, as the rules will be triggered simultaneously and will try to change the state of the living room lights to different states.

RQ5. *Is it possible to build a system for a smart home context capable of predicting **conflicts caused by redundant rules**?*

Two rules are redundant when both define the same triggers and simultaneously trigger the same action.

Take attention to the following example:

- Rule 1: If the outside temperature is lower than 10°C, turn on the heater.
- Rule 2: If the outside temperature is lower than 15°C, turn on the heater.

Here, Rule 1 and 2 are redundant, as when Rule 1 is triggered, Rule 2 is also triggered. Rule 1 can be deleted in this case, as its effects are already contained in Rule 2.

RQ6. *Is it possible to build a system for a smart home context capable of predicting different conflict types and **takes into account each rule periodicity** when deciding if a set of rules contains conflicts?*

Rules have the same periodicity if, at some point in time, it is possible for them to execute simultaneously.

RQ7. *Is it possible to build a system for a smart home context capable of **providing explanations about the existing conflicts** to the users so that they can easily understand why a certain conflict occurs?*

By explaining why a conflict occurs, users will have access to a concrete scenario in which a set of rules conflicts, helping to understand why the rules conflict.

4.5 Methodology

We decided to develop a simple system with easily definable and analyzable rules to validate the abovementioned hypothesis. Rather than utilizing a complex and mature IoT system such as Node-RED¹, the choice was made to create a simple system that would enable rapid prototyping and reconfiguration with minimal complications. This approach offers several advantages for validating the hypothesis. Firstly, developing a simple system allows for a more focused analysis, making it easier to observe and understand the behavior of the graph-based approach. Secondly,

¹Node-RED, <https://nodered.org/>

the ability to prototype and reconfigure the system quickly facilitates iterative testing and refinement of the approach, leading to more efficient hypothesis validation. Overall, opting for a simple system enables a streamlined and agile validation process while maintaining the flexibility necessary for experimentation and adaptation.

We can divide this process into the following steps:

- Generate an initial set of rules without having conflicts between them and then generate sets of rules containing loops, redundancies, and inconsistencies with the initially generated rules (one set for each type of conflict);
- Verify if the system correctly predicts conflicts;
- Perform an extensive analysis to know concretely all the situations in which our system can detect redundancies and inconsistencies;
- Statistical analysis to compare whether the implementation of the connected components algorithm resulted in an effective improvement in our system's efficiency.

4.6 Assumptions

We have made an assumption to simplify the process of implementing our system and validating the hypothesis. Although a real IoT system usually involves multiple vendor-specific or technology-specific control applications, for this work, we consider that we can centralize all the automation rules in a single control plane.

4.7 Summary

This chapter starts with an exposure of the main issues identified (*cf.* Section 4.1, p. 32) while we reviewed the literature in Chapter 3 (p. 8). In Section 4.2 (p. 33), we define our *desiderata*, presenting the main goals and objectives we want to achieve in this work ideally. After that, we present our hypothesis and a more in-depth explanation specifying which system we want to build and which path to implementing it in Section 4.3 (p. 35). To guide our research and help to validate the hypothesis, we created some research questions in Section 4.4 (p. 35). Then we defined the methodology we will follow for validation in Section 4.5 (p. 37), along with an assumption that persists throughout the subsequent chapters.

Chapter 5

Implementation

5.1 Overview	39
5.2 Implementation Details	40
5.2.1 Rules Definition	40
5.2.2 Get Graph Valid Scenarios and Graph Generation	42
5.2.3 Rules Parsing and Edges Creation	45
5.2.4 Check Periodicity Intersection Between Two Rules	46
5.2.5 Conflict Detection in Rules	47
5.2.6 Export graphs to GEXF format	56
5.3 Known Limitations	58
5.4 Summary	59

This chapter details implementing our solution to solve the problem stated in Chapter 4 (p. 32). We supply information about the internal functioning of our system, its architecture that makes it possible to detect various types of conflicts and other details that we deem essential to fully understanding our solution. Section 5.1 (p. 39) delivers an overview of the designed solution, whose implementation details are further explored in Section 5.2 (p. 40). We explore the limitations of our implementation and its impact on our system in Section 5.3 (p. 58), and, finally, we conclude this chapter with a summary in Section 5.4 (p. 59).

5.1 Overview

In this work, we carry out a system capable of detecting conflicts inside a set of trigger-action rules defined for a smart environment and explaining them to the users easily. This solution aims to facilitate the rules' creation process and subsequent verification of conflicts between them so that unexpected or unwanted events do not disturb the users' daily lives.

Consequently, we developed an innovative approach for conflict recognition based on directed graphs. Each graph vertex represents a possible scenario for existing devices in a smart environment, and each edge symbolizes a rule, connecting a vertex with the initial state and another with

the final state after the rule is applied. As there can be rules that connect the same vertices — rules triggered in the same circumstances —, then these directed graphs are also multigraphs. Starting from here and using the graphs' particularities, we identified the various conflicts we proposed in Section 4.2 (p. 33).

The final result of our work allows users to submit a YAML file where detectors, actuators, and rules are defined and then receive as an answer the pairs or sets of conflicting rules and a small explanation of why it is a conflict.

5.2 Implementation Details

Throughout this section, we will explore and explain the most relevant implementation details, as well as clarify why we made certain decisions over others that were also correct.

We used JavaScript as the programming language to develop our solution because it is an easy-to-use language, and its versatility and interoperability make it easy to integrate into a more appealing graphical interface for users in the future.

5.2.1 Rules Definition

To use our system, users must first define all the detectors and actuators used in a YAML file and then specify the rules for a given smart space.

```
1 detectors:
2   - id: motion_living_room_sensor
3     type: sensor
4     valueType: boolean
5     values: [true, false]
6
7 actuators:
8   - id: lights_living_room
9     type: lights
10    valueType: numeric interval
11    range: [1, 100]
12
13 rules:
14   - name: rule_1
15     description: 'If motion is detected in the living room, set the living room
16               lights' brightness to 50%. (Always)'
17     triggers:
18       trigger1:
19         detector_id: motion_living_room_sensor
20         condition:
21           comparison: '='
22           value: true
23     action:
24       actuator_id: lights_living_room
```

```

24     arguments:
25         command: 50
26     periodicity: '* * * * *'
```

Listing 5.1: Example of a detector, an actuator and a rule definition in YAML

We can divide the file into three parts:

- **Detectors part:** where the existent detectors in the smart environment are defined;
- **Actuators part:** definition of the existing actuators in the environment ;
- **Rules part:** rules definition using the detectors previously defined for the triggers and actuators for the action part.

As shown above in Listing 5.1 (p. 40), both detectors' and actuators' characterization parts are similar. In both cases, an *id* is assigned to each device, as well as its *type* (e.g., lights, sensor, door, window, among others). In case some sensors and actuators are related (e.g., door and the respective sensor), the *id* of both must be the same, adding “_sensor” at the end in the case of the sensor. A *valueType* is also defined — boolean or numeric interval —, considering the possible values the device can assume. Depending on the previous value, the following tag can assume two different names: (i) *range*, in case of a numeric interval, (ii) *values*, in case of boolean. In the first case, the values entered correspond to the range of values the device works. Instead, in the second case, the values introduced are the specific states that the device can assume.

Looking now at the rules part, it starts by defining a name and briefly describing the rule's purpose. Then comes the trigger's definition part. That can be empty if the rule has no trigger, or it can have one or more triggers defined (assuming a conjunction between them), each with a different name. Inside each trigger, users select a detector, and then they define the condition that the detector must obey. The *comparison* value can be $=$, $\neq$, $<$, $>$, $\leq$, or $\geq$, while the *value* must be one of the previously defined valid values for the selected device. Moving forward to the action definition, users have to choose an actuator. Then a valid command is provided, which must always follow the previously defined values. Each rule can have zero, one or more triggers, but one and only one action. With this approach, defining two simultaneous commands for the same actuator is impossible. For example, the action “turn on the HVAC and set its temperature to 22°C” is invalid. Our implementation only accepts the second part of this action. Lastly, there is the rule's periodicity definition through a cron expression, an expression in string format that defines schedules. This type of expression allows one to specify the timing and frequency at which a particular task or job should be executed (cf. Listing 5.2).

```

1 * * * * *
2 | | | | |
3 | | | | +-- Day of the Week (0-7, Sunday is 0 or 7)
4 | | | +---- Month (1-12)
```

```

5 | | +----- Day of the Month (1-31)
6 | +----- Hour (0-23)
7 +----- Minute (0-59)
8
9 * means all values
10 - used for ranges
11 , used to specify multiple values
12 / used for increments

```

Listing 5.2: Cron expression general format and values

Parsing this file gives rise to three arrays of dictionaries: one contains the detectors, another contains the actuators, and the third contains the rules. After reading the rules from the YAML file, we filter them in order to eliminate rules that exist in duplicate, that is, repeated rules. We consider two rules equal when their triggers, actions, and periodicity are the same.

5.2.2 Get Graph Valid Scenarios and Graph Generation

Once the smart home devices and rules are defined, the system has to create all possible valid scenarios for the smart environment. The system then assigns each scenario to a vertex of the graph.

Regarding the states that each device can assume, we saw in Section 5.2.1 (p. 40) that in case its *valueType* is *boolean*, the possible states are immediately provided. However, if *valueType* is *numeric interval*, the device definition in the YAML file only provides the operating interval. In this case, it is necessary to divide this interval into several small intervals to obtain the various states where the device can be found. For the system to get these states, there are two main strategies:

- dividing the interval unitarily, with each possible state being one of these unit intervals. For example, if we have a device with a range [1, 100], we would obtain the states [1, 2], [2, 3] up to the state [99,100].
- for each device of type *numeric interval*, go through the rules and check which values are assigned to it. Then, sort the values and obtain the corresponding states. As an example, if we have a sensor with a range [1,100] that is assigned the condition “less than 20” in one trigger and the condition “greater than or equal to 55” in another, the corresponding states are [1, 19], [20, 54] and [55, 100].

As we can easily conclude, the second option is better than the first because it allows us to have significantly fewer states in cases of large intervals. Furthermore, this option will greatly simplify the following stages of our implementation, as it facilitates the attribution of valid states for each rule and reduces the size of the final graph. To the generated states, we add another one, “off”, to indicate that the device is offline.

We consider a scenario possible and valid if all the devices are in one of the states previously defined for each one and the relations between the states of the devices are coherent. To clarify this last part of the definition, consider an environment with two gadgets: a lights sensor (that can be active or off) and the respective lights (that can be on or off) to which this sensor refers. In this case, it does not make sense to have a scenario where the lights sensor is active, detecting the lights as being on, and the lights state is off. This combination is invalid. Only scenarios where lights and the respective sensor are in the same state are valid.

After generating all possible scenarios and attributing each one to a different vertex of the directed graph, we started to notice a potential problem that later on, in a more advanced phase of our work, would significantly decrease the efficiency of our solution: as the number of devices increases, the number of possible scenarios grows exponentially (*e.g.*, 2^n , 3^n (considering n the number of devices), depending on the number of states of each device). This exponential growth is not a very relevant problem in a scenario with few devices and few rules (*e.g.*, 10, 12), as the number of vertexes and edges is relatively small and does not imply a great complexity in temporal and spatial terms. However, let us consider an environment with more devices and rules (*e.g.*, 40, 50). This scenario already makes it much more complicated because we will have thousands of millions of scenarios and too many edges connecting them as, possibly, with more existing scenarios, there will be a more significant number of relationships between them for each rule. Searching for a state or comparing all outgoing edges from each node will be much more complex. In this situation, our solution would become very time and space-consuming and, therefore, unfeasible for use in larger spaces.

There is a tendency for groups of devices to form that relate to each other and are involved in the same rules (strongly connected components) in a smart environment. It is unusual, for example, for the room temperature to influence the current state of the garage door. On the other hand, there is a greater possibility of a relationship between the outside temperature and the state of the HVAC equipment in the bedroom. In this way, it will not be necessary to consider the garage door and the outside temperature in the same graph. The possible state combinations for these devices will only increase the number of possible scenarios for the smart environment. Thus, if we manage to isolate sets of devices, creating a graph associated with each set, reducing the number of possible scenarios will be possible since the existing devices in a graph only relate to each other and not to the other existing devices in the remaining graphs.

Therefore, to overcome this problem, we decided to resort to the connected components algorithm that can help us to reduce the total number of scenarios. This algorithm aims to identify disjoint subsets, known as connected components, within an undirected graph. In simple terms, it helps determine groups of nodes that are interconnected or can reach each other through a series of edges. To apply it to our case, firstly, we must discover which devices are related to each other before obtaining all possible scenarios. After that, instead of having just one graph with all the devices associated with it, we will have several smaller graphs, each one containing only the devices that relate to each other.

Starting from the existing information in the YAML file, we built an undirected graph in which

each vertex corresponds to one of the existing devices (sensors and actuators), and each edge represents a relationship between the devices. Two devices are related in case they are used in the same rule. If the device is a sensor, and there is an actuator related to it or vice versa (*e.g.*, lights sensor and the respective lights) that is not mentioned in the rule, the system also creates an edge between them. Then, having generated the graph, the rest of the algorithm is based on iterating through each of the nodes of the graph and, if it has not yet been visited, starting a depth-first search (it could also be a breath-first search), visiting all the nodes that are possible to achieve from that. When the search ends (it is impossible to visit any more nodes), we have a set of connected devices. This process is repeated starting from another node that has yet to be visited until all the graph nodes are visited. In the end, we get several disjoint subsets of interconnected nodes. From these subsets, and using the previously described approach, we obtain for each one of them all the possible scenarios for the devices belonging to them, associating each one to a different vertex of the graph.

Applying the connected components algorithm to our solution before generating all the possible states allows us to reduce the temporal and spatial complexity of the code, as finding a vertex with a particular state is easier and faster when the search space is smaller. It is what happens when we apply the connected components algorithm. Although this way, we have more graphs than before, they are smaller, and we precisely know which one we have to scour for a specific state — we have a *Map* where we associate each graph id with the devices on that, so we do not need to be traversing all the graphs.

Let us examine the following example to understand what has been described and the differences between the two approaches.

Example Scenario Consider a smart environment containing the following devices (and the respective states) and rules.

Devices and states:

- motion sensor: motion, no motion
- blinds sensor: opened, closed
- door sensor: opened, closed
- lights: on, off
- TV: on, off

Rules:

- Rule 1: If motion is detected, turn on the lights.
- Rule 2: If blinds are opened, turn off the lights.
- Rule 3: If the door is closed, turn on the TV.

Considering our initial approach, we will have 2^5 (32) vertexes in the graph. In Figure 5.1 (p. 45), we represent part of the resulting vertices.

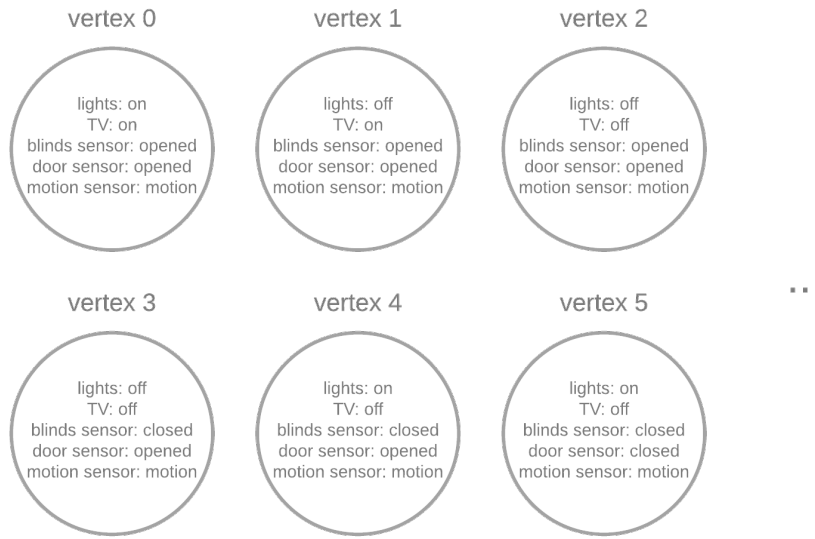


Figure 5.1: Part of the resultant vertexes without using connected components algorithm

Nevertheless, suppose we apply the connected components algorithm before getting all valid scenarios. In that case, we will obtain two disjoint subsets: one containing the lights, the motion sensor, and the windows sensor, and the other having the TV and the door sensor. Rules 1 and 2 relate the devices from the first subset, while Rule 3 relates the other two devices. Thus, the number of vertexes in the graph will be reduced to $2^3 + 2^2$ (12). We represent the resulting scenarios in Figure 5.2.



Figure 5.2: Resultant vertexes using connected components algorithm

5.2.3 Rules Parsing and Edges Creation

After spawning all the vertices of the various graphs, it is time to examine each rule and create the matching edges in the graphs.

In our solution, we must get the graph containing the devices associated with the rule and then obtain the vertexes whose associated states are valid initial states for the rule under analysis. A state is a valid initial state if the value assigned to each device referred to in the rule's triggers respects the condition presented there. In addition, the value defined for the actuator in this state must differ from the final value it assumes after the rule's execution. In other words, if a rule states in its trigger that a sensor must be in *state1* and that if this happens, then the actuator state should change to *stateB*, the initial valid scenarios are the ones where the sensor state is *state1* and the actuator state is different from *stateB*. If there are no triggers in the rule, we only have to attend to the actuator's state. Thus, we regard the states in which the actuator state differs from the final state intended by the rule as the valid initial states.

With the initial vertex found, we need to find the vertex containing the final scenario. Considering that when a rule is triggered, none else is triggered simultaneously, the only device that will change its state with the rule execution is the actuator. Sometimes, if the actuator has an associated sensor (e.g., lights and the respective lights sensor), the state of that sensor will also change. Thus, for each initial scenario found, the respective final scenario after the execution of the rule is equal to that scenario, changing only the value of the actuator (and of the respective sensor, if any, in some cases), which becomes the value defined in the action of the rule. Straightaway, we only have to search for the final scenario in the graph and then create the respective edge connecting the initial and final vertexes.

With this implementation, the final graphs are multigraphs, *i.e.*, in the final graphs (after creating all the edges), there may be parallel edges — multiple edges that connect the same pair of nodes. These parallel edges are justified because there may be different rules with the same initial and final states. Another critical point in our implementation is that there are no edges whose initial and final vertices are the same. As we said before, the actuator value will always differ from the intended final value in the initial state. Therefore, when the rule is triggered and the actuator is already in the desired state, there is no rule execution, which impedes the existence of self-pointing edges, *i.e.*, self-loops.

5.2.4 Check Periodicity Intersection Between Two Rules

When we are analyzing a pair of rules to know if there could be any conflict between them, it is also necessary to look at the periodicity of each one. Rules will run concurrently just in case their periodicities overlap or intersect in the future. We implement a function to check periodicities intersection to determine if this happens. This function takes as arguments two cron expressions. It starts by checking if the expressions are the same or if any of them is equal to the string “* * * * *”, which corresponds to the periodicity *Always*. There is still a third hypothesis: we resort to the *parser* module of the *cron-parser*¹ package to perform a more detailed check. It examines each field (minute, hour, day of month, month, and day of week) in *periodicity1* and *periodicity2* and checks if there is any overlap in their values. For each field, the code uses the *some* method

¹Cron-parser package, <https://www.npmjs.com/package/cron-parser>

to iterate over the elements of *parse1.fields.fieldName* and checks if there is any element that is also present in *parse2.fields.fieldName*. If at least one common element exists for each field, there is an intersection between the two periodicities. If one of these three hypotheses is verified, the function returns *true*. If none of those conditions are satisfied, the function returns *false*, indicating no intersection between the two periodicities.

We had initially implemented this function differently. The main difference is that in this initial implementation, we used the *parseExpression()* function of the *parser* module and then calculated all the moments in the future that correspond to each of the cron expressions. Then, to know if the expressions intersected, it was enough to check if any of the values of one of the dates lists existed in the other. However, we abandoned this implementation because it was much more time-consuming than the current one. Calculating dates corresponding to a particular expression and searching them in the other expression dates list takes a long time, not forgetting that the calculation must be performed twice, once for each rule of the pair. As this process will be repeated countless times for a set of rules, it becomes impracticable to use this implementation.

5.2.5 Conflict Detection in Rules

As we mention in Section 4.2 (p. 33), we want to address three conflict types with this work:

- Loop conflicts;
- Conflicts between rules with (partially) the same trigger but different actions in the same device;
- Conflicts caused by redundant rules.

To detect all these types of conflicts, we implement, taking advantage of the properties of directed graphs, the algorithms we will present in the following sections.

5.2.5.1 Loop Conflicts

The first conflict type we are going to discuss is the loop conflict. A set of rules constantly activated without reaching a stable final state causes this kind of conflict. These rules will continuously execute, with an uninterrupted change in the states where the involved devices are found. Thus, based on previous definitions [9, 18], in a smart environment, we can define a *loop* as a situation where a collection of rules repeatedly cyclically trigger each other, creating an infinite loop of actions or events. There is a cyclical dependency across the rules. Loops can result in undesirable or unexpected behavior within the smart environment.

In our solution, we assume that there must be at least three conflicting rules for a loop to exist. In terms of the generated graph, a loop comprises at least three edges (which represent the rules) and four vertices (three unique ones, with the initial and final ones being the same). Although there may be conflicts between only two rules, we consider that in this case, we are facing another type of conflict — inconsistencies (*e.g.*, same triggers, different actions on the same device; different

triggers, different actions on the same device). Section 5.2.5.2 (p. 50) addresses the resolution of part of this conflict type.

As we base our implementation on directed graphs, finding the existing loops in each is relatively simple. Resorting again to a depth-first search (DFS), we implemented an algorithm capable of detecting the extant loops in a graph, regardless of their size. That is one of the advantages of our graph-based implementation, as it allows us to find n-size loops easily.

Algorithm 1: *hasLoop()* Algorithm

output: *loops*, an array of dictionaries containing the loops' information

```

visited  $\leftarrow$  newSet();
stack  $\leftarrow$  newSet();
loops  $\leftarrow$  [];
visitedPaths  $\leftarrow$  [];

for every vertex in vertices list do
    if visited not contains vertex.id then
        | dfs(vertex, visited, stack, loops, visitedPaths);
    end
end

```

The first step involves implementing the *hasLoop()* function (cf. Algorithm 1), where four data structures — *visited*, *stack*, *loops*, and *visitedPaths* — are initialized. It is in these structures that information about the visited vertices and the founded loops will be stored. These data will be helpful for the next phase, *i.e.*, the DFS traversal. The code then iterates over all vertices in the graph using a *for...of* loop and checks if the vertex has been visited. In case not, it calls the *dfs()* function (explained later) to perform a DFS traversal starting from that vertex. Finally, after all the vertices have been visited and the DFS traversal is complete, the method returns the *loops* array containing information about any detected loops in the graph.

The *dfs()* function is a recursive function that performs a depth-first search traversal from a given vertex (cf. Appendix A, p. 107). It takes five parameters as arguments:

- *vertex*: the current vertex being explored;
- *visited*: a *Set* of visited nodes;
- *stack*: a *Set* representing the current path in the DFS traversal;
- *loops*: an array to store the information about detected loops;
- *visitedPaths*: an array to keep track of visited paths.

Each time the system calls *dfs()* function, it adds the current vertex to the two previously mentioned sets. Then, we iterate through every outgoing edge from the current vertex and check if we have already visited the corresponding end vertex, *i.e.*, if it is in the *stack*. If so, then we find a loop. When a loop is detected, the algorithm creates a *loopPath* array by extracting the vertices

from the *stack* starting from the index where the loop was found until the end of the set. The loop vertex is added again at the end to complete the loop. Then we append the loop information to the *loops* array in the dictionary format. This dictionary contains two keys: (i) *vertexIds*, containing the ids of the vertices that belong to the loop; (ii) *associatedRules*, that holds the rules associated with the loop. To obtain the rules, we need to discover, for each consecutive pair of vertices existing in the *loopPath*, the rule associated with the edge that connects them. Nevertheless, if we have not visited the end vertex, we recursively call the *dfs()* function to continue the traversal from that vertex. After exploring all outgoing edges from the current vertex, it removes the current vertex from the *stack* and *visited* sets. This step allows revisiting the vertex in different paths.

We need to call the *hasLoop()* function for each directed graph we created earlier to get all the extant loops in a set of rules. After that, we have to check, for each loop found, whether the associated rules can be triggered simultaneously, *i.e.*, if their periodicities intersect at any moment. In case this condition is true, then this is a valid loop.

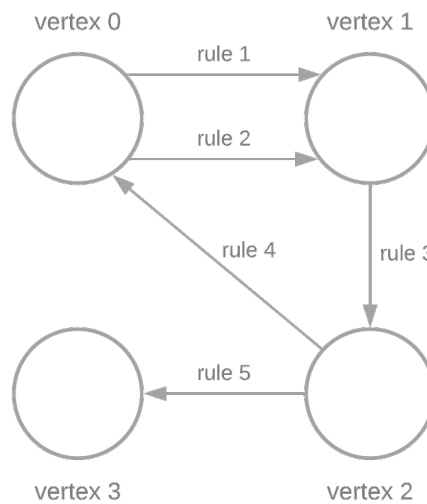


Figure 5.3: Example of a graph with parallel edges that contains a loop

As we previously mentioned, in our implementation, there is the possibility that the graphs have parallel edges (multigraph). Therefore, this allows the same set of vertices can be associated with several rule combinations in a loop. Considering the graph in Figure 5.3, there is a loop between vertices *vertex 0*, *vertex 1*, and *vertex 2*. This loop generates two different valid paths of conflicting rules due to the parallel edges between *vertex 0* and *vertex 1* — *rule 1*, *rule 3* and *rule 4*; *rule 2*, *rule 3* and *rule 4*. In implementing the *dfs()* function, we chose to look for only one of the possible combinations. We think that this simplification makes identifying loops easier for the user and may also facilitate their correction. As we saw earlier [9], loops are considered the most challenging issues to understand. Consequently, providing a massive list of existing loops in a set of rules, in which one or more rules can change from one combination to another, could

further complicate this already complex task. Suppose the user wants to recheck the existence of conflicts after correcting previously existing loops. If another combination of rules still exists for a particular loop, we will make it known this time.

Returning to Appendix A (p. 107), we see that when we discover a new path of a possible loop, there is a second condition *else if*. We intend this condition for loops formed by two rules and will explore it in Section 5.2.5.2, as we believe they are inconsistencies and not loops.

5.2.5.2 Conflicts Between Rules with (partially) the Same Trigger but Different Actions in the Same Device

Another type of conflict we want to detect in our implementation is inconsistencies. Let us start by looking at the following examples:

Example 1:

- **Rule 1:** When the garage door is opened, turn on the garage lights and set the brightness to 35%.
- **Rule 2:** When the garage door is opened, turn on the garage lights and set the brightness to 80%.

Example 2:

- **Rule 1:** If motion is detected in the living room, turn off the living room lights.
- **Rule 2:** If the living room door is opened, turn on the living room lights and set the brightness to 63%.

These two examples represent the main types of existing inconsistencies. In *Example 1*, we have two rules with the same trigger — “When the garage door is opened” —, but different actions in the same actuator — “turn on the garage lights and set the brightness to 35%” vs. “turn on the garage lights and set the brightness to 80%”. Both these rules are always triggered simultaneously, but having different actions will cause different states for the same actuator. Alternately, in *Example 2*, we have two rules with different triggers — “If motion is detected in the living room” vs. “If the living room door is opened” — and actions, but actuate in the same actuator — “turn off the living room lights” vs. “turn on the living room lights and set the brightness to 63%”. These rules will only be executed simultaneously when both triggers are valid, *i.e.*, in a scenario where motion is detected in the living room and the living room door is opened. A conflict will arise as we have different actions for the same device.

Thus, in a smart environment, we can define an *inconsistency conflict* in trigger-action rules as a situation where rules triggered (almost) simultaneously have conflicting or contradictory actions, resulting in multiple states for the same device. We can divide inconsistencies into two types: (i) Conflicts between rules with the same triggers but different actions in the same device; (ii) Conflicts between rules with different triggers and actions but actuate in the same device. In our implementation, we only focus on inconsistencies of the first type.

Focusing now on the type of conflict that interests us, we have to take into account the various possibilities of combinations between the triggers of the two rules that can occur:

- Rules share the same sensors: all sensors used in the triggers of a rule are also used in the triggers of the other rule;
- Rules share only part of the sensors: some of the sensors used in the triggers of one rule are also used in the triggers of the other rule;
- One of the rules has no trigger part;
- Both rules have no trigger part.

Given that sensors can have two types — *boolean* or *numeric* —, we also have to pay attention to this attribute, being certain that if the device is the same, then its type must be the same: (i) *boolean* vs. *boolean*; (ii) *numeric* vs. *numeric*. We can use the following symbols to compare the sensor's current value and the value we define in the trigger: $=$, $\neq$, $<$, $>$, $\leq$, $\geq$. Lastly, it is also necessary to compare the values assigned to the sensor: (i) $value1 < value2$; (ii) $value2 < value1$; (iii) $value1 = value2$; (iv) $value1 \neq value2$. We must consider these four factors to check if two rules have (partially) the same triggers.

Starting with the first two rule sensors' combinations mentioned — rules sharing the same sensors and rules sharing only part of the sensors —, two triggers using the same sensor will be similar if they fulfill any existing combinations in Table 5.1 (p. 52). We do not need to analyze the remaining combinations when one or both rules have no trigger — these combinations do not exist in one or both rules. We can examine the actions after analyzing the triggers' part of the rules.

Remembering the structure of the action of a rule in the YAML file (*cf.* Listing 5.1, p. 40), within each action, an actuator (*actuator_id*) is chosen in which a *command* will be executed. For two rules to have the same action, they must act on the same device — same *actuator_id* — and the selected command must be the same in both actions — same *command*. If this scenario occurs, it remains only to check whether the two rules can be executed at the same time, that is, whether their periodicities intersect at some point in time. If so, then we found two conflicting rules. The valid possible combinations for rules' periodicity are the following:

- Always vs. Always;
- Always vs. Specific Time (*e.g.*, “Daily at 16:30”);
- Always vs. Time Interval (*e.g.*, “Every Tuesday between 16:00 and 18:00”);
- Specific Time vs. Specific Time, both the same;
- Specific Time vs. Time Interval, the interval contains the specific time;
- Time Interval vs. Time Interval completely intersect each other;
- Time Interval vs. Time Interval partially intersect each other;

Sensors Type	Trigger1 Symbol	Trigger2 Symbol	Symbol Restrictions	Values Comparison
boolean vs. boolean	$=, \neq$	$=, \neq$	$symbol1 = symbol2$	$value1 = value2$
	$=, \neq$	$=, \neq$	$symbol1 \neq symbol2$	$value1 \neq value2$
numeric vs. numeric	$<, \leq, >, \geq$	$<, \leq, >, \geq$	$symbol1 = symbol2$	$value1 < value2$
				$value2 < value1$
				$value1 = value2$
				$value1 < value2$
	$<, \leq$	$<, \leq$	$symbol1 \neq symbol2$	$value2 < value1$
				$value1 = value2$
				$value1 < value2$
				$value2 < value1$
	$>, \geq$	$>, \geq$	$symbol1 \neq symbol2$	$value2 < value1$
				$value1 = value2$
	$<, =, \leq$	$>, =, \geq$	$symbol1 \neq symbol2$	$value2 < value1$
				$value2 < value1$
				$value1 = value2$
				$value1 < value2$
	$=, \geq$	$\leq, =$	$symbol1 \neq symbol2$	$value1 = value2$
				$value1 < value2$
				$value1 < value2$
				$value1 < value2$
	$>, \geq, =$	$<, \leq, =$	$symbol1 \neq symbol2$	$value1 < value2$
				$value1 \neq value2$
				$value1 \neq value2$
				$value1 = value2$

Table 5.1: Possible combinations to compare two triggers using the same sensor to detect inconsistencies

- Time Interval vs. Time Interval one contains the other.

If two rules share sensors in their triggers (or do not have a trigger), they will be executed in at least some common scenarios. Thus, translating this to what happens in a directed graph, two rules with the same trigger but different actions on the same actuator are represented through edges that start from the same initial node but are directed to different final vertices. However, more than this condition is required to discover all pairs of rules with this type of conflict. As mentioned in Section 5.2.5.1 (p. 47), there are pairs of rules containing inconsistencies detected through loops and not using the previously described method. This alternative detection happens as there are cases in which the action of a rule is associated with the trigger of the other.

In the case of *Example 1*, consider an initial scenario where the garage door is open, and the lights are off. In this case, both rules will execute simultaneously, creating a conflicting situation because Rule 1 sets the brightness of the lights to 35% while Rule 2 sets the brightness of the same lights to 80%. This conflict in the graph is represented through two edges that come out of the same node (representative of the initial scenario) but with different final directions. The edge associated with Rule 1 goes to the vertex that represents the scenario where the brightness of the garage lights is at 35%. In contrast, the edge associated with Rule 2 goes to the vertex that represents the scenario where the brightness of the lights is at 80%.

Let us now consider another example:

Example 3:

- **Rule 1:** If the living room windows are opened, close the living room curtains.
- **Rule 2:** If the living room windows are opened, open the living room curtains.

In this example, the graph will no longer represent the conflict like the previous one. Considering an initial scenario where the living room windows are open, and the curtains are closed, only Rule 2 will execute. Rule 1 is not initially executed because the curtains are already in the desired state in the initial scenario. However, after running Rule 2, the curtains in the living room open, which allows running Rule 1. Running the first rule, we return to the initial scenario, and Rule 2 will run again, entering here in a cycle. This way, the graph will detect this conflict as a cycle between two vertices representing the two possible scenarios. Although the graph represents this case as a cycle between two vertices, we consider this conflict an inconsistency rather than a loop. We made this decision as the two rules have the same trigger but different actions on the same actuator, which fits the inconsistencies conflict.

To make it possible to get every pair of conflicting rules with the same triggers but different actions in the same device, we have to apply the previously explained checks to all our directed graphs. We divide this process into two parts.

More concretely, firstly, we have to check, for each pair of rules that come out of the same vertex (pair of outgoing edges), if they share the same sensors, part of them or do not have triggers, if they have the same action, and if the periodicities intersect. Algorithm 2 briefly describes the algorithm we implemented to detect this type of conflict when rules share the same initial scenario.

The second phase of this process involves getting the loops formed by two rules obtained using the functions *hasLoop()* and *dfs()*. If we retrieve the Appendix A (p. 107), the loops of two rules are discovered inside the *else if* condition after discovering a new path for a possible loop. Unlike our decision for loops with three or more rules, we present the user with all possible rule combinations for a given path. After obtaining the loops between two rules, it is necessary to distinguish which of these loops refer to cases of rules with the same trigger but different actions on the same actuator from cases that concern conflicts between rules with different triggers and actions that act on the same device. As we have already analyzed, we consider that the rules have similar triggers when they share at least one of the sensors or when at least one of them does not have a trigger part. This consideration is precisely the verification that the system needs to do to choose the cases of inconsistencies that interest us. As mentioned before, inconsistencies between rules that have different triggers and actions but act on the same actuator are not part of the scope of our project.

When the algorithm finds a conflict, it presents the users with a small explanation about why there is a conflict. It identifies the rules in conflict and describes a possible initial state for the used sensor(s) and what are the conflicting actions. This way, we make it easier for users to identify the conflict, describing a possible situation in which it will occur. Returning to the initial *Example 1*, the user would be given the following information: “Rule 1 and Rule 2 are in conflict: When *inside_door_garage_sensor* is open, set brightness to 35% of *lights_garage* and set brightness to 80% of *lights_garage* are contradictory actions.”

Algorithm 2: Algorithm to discover conflicts between rules with the same triggers but different actions in the same device

```

for every graph in graphs list do
    vertices  $\leftarrow$  graph.getVertices;
    rules_found  $\leftarrow$  [];
    for every vertex in vertices do
        edges  $\leftarrow$  vertex.getEdges;
        for  $i \leftarrow 0$  to edges.lenght - 1 do
            rule1  $\leftarrow$  edges[i].getAssociatedRule;
            for  $j \leftarrow i + 1$  to edges.lenght do
                rule2  $\leftarrow$  edges[j].getAssociatedRule;
                res  $\leftarrow$  haveSameTriggers(rule1, rule2);
                if res then
                    same_action_actuator  $\leftarrow$  rule1.action.actuator_id =
                        rule2.action.actuator_id;
                    diff_action_command  $\leftarrow$  rule1.action.arguments.command  $\neq$ 
                        rule2.action.arguments.command;
                    same_periodicity  $\leftarrow$ 
                        checkPeriodicityIntersection(rule1.periodicity, rule2.periodicity);
                    new_rule  $\leftarrow$  not rules_found contains pair (rule1, rule2);
                    if same_action_actuator & diff_action_command & same_periodicity
                        & new_rule then
                        | append pair (rule1, rule2) to rules_found;
                    end
                end
            end
        end
    end
end

```

5.2.5.3 Conflicts Caused by Redundant Rules

Redundancies are the last conflict type we want to detect in our project. Two rules are redundant when they have overlapping or identical conditions and actions. Concretely, two rules are redundant when both rules define the same trigger and simultaneously trigger the same action. Redundant rules are nothing new regarding functionality or value to the system, as they do the same thing simultaneously. They only make the system less efficient, forcing it to allocate resources to process something unnecessary and repeated.

Example 1:

- **Rule 1:** If the bedroom temperature falls below 18°C, set the bedroom HVAC to 20°C.
- **Rule 2:** If the bedroom temperature falls below 15°C, set the bedroom HVAC to 20°C.

Both rules contained in *Example 1* are triggered at the same time in case the bedroom temperature is below 15°C. Their triggers overlap as the temperature below 15°C necessarily implies that it is below 18°C. This pair of rules has the same action — “set the bedroom HVAC to 20°C” —, so the rules are redundant. *Rule 2* is unnecessary for the system. Having only *Rule 1* would suffice and work the same way.

The procedure for identifying this type of conflict in our system is similar to the one we use for the conflict dealt with in Section 5.2.5.2 (p. 50). We have to consider the possible combinations between the triggers of the two rules, the sensors’ type, the comparison symbol used in each trigger, and the comparison between the values assigned to the sensor in each trigger.

Starting with the valid combinations between the triggers of both rules, for that conflict type, we have: (i) rules sharing the same sensors; (ii) both rules have no trigger part. In this case, the remaining combinations — rules sharing only part of the sensors and one of the rules has no trigger part — do not apply to redundancies as, in these cases, the triggers no longer overlap in every situation. Using the above definition of redundancy, we conclude that it is necessary that the two rules use the same devices or do not have triggers. In any other case, the rules could no longer be triggered simultaneously in all situations — conflicts analyzed in Section 5.2.5.2 (p. 50). As our system only allows the use of boolean or numeric sensors, the two possible combinations for sensor types are: (i) *boolean vs. boolean*; (ii) *numeric vs. numeric*. We can use the following symbols to compare the sensor’s current value and the value we define in the trigger: $=$, $\neq$, $<$, $>$, $\leq$, $\geq$. Lastly, to compare the values assigned to the sensor, we have the same possibilities that we mentioned in the previous conflict: (i) $value1 < value2$; (ii) $value2 < value1$; (iii) $value1 = value2$; (iv) $value1 \neq value2$. We must consider all these conditions when deciding whether two possibly redundant rules have overlapping triggers.

In Table 5.2 (p. 56), we focus on rules sharing the same sensors and the valid combinations for the remaining parameters that allow us to assert that two triggers coincide. In case both rules have no trigger part, it is not necessary to analyze the remaining conditions (they do not exist), remembering that in this phase, there will only be redundant rules without triggers in the system

Sensors Type	Trigger1 Symbol	Trigger2 Symbol	Symbol Restrictions	Values Comparison
boolean vs. boolean	$=, \neq$	$=, \neq$	$symbol1 \neq symbol2$	$value1 \neq value2$
	$=, \neq$	$=, \neq$	$symbol1 = symbol2$	$value1 = value2$
numeric vs. numeric	$<, \leq, >, \geq$	$<, \leq, >, \geq$	$symbol1 = symbol2$	$value1 < value2$
				$value2 < value1$
	$<, \leq$	$<, \leq$	$symbol1 \neq symbol2$	$value1 = value2$
				$value1 < value2$
	$>, \geq$	$>, \geq$	$symbol1 \neq symbol2$	$value2 < value1$
				$value1 = value2$
	$=, \neq$	$=, \neq$	$symbol1 = symbol2$	$value1 = value2$

Table 5.2: Possible combinations to compare two triggers using the same sensor to detect redundancies

if they have different periodicities. As we saw in Section 5.2.1 (p. 40) if two rules have the exact same trigger, the same action, and the same periodicity, one of them is deleted.

If the triggers fit in any possible combinations, the next step is to verify if the pair of rules has the same action, *i.e.*, if it acts on the same actuator with the same command. If so, it only remains to check whether the periodicities of the rules intersect at some point in time, which in turn allows us to state that the two rules are redundant.

Again, as we have several graphs, we need to apply the algorithm just explained to all vertices of each graph to obtain all pairs of redundant rules. For each pair of rules that come out of the same vertex, it is necessary to check if their triggers overlap (or do not exist), if the respective actions are equal, and if their periodicities intersect at some point — Algorithm 3. When the algorithm finds a pair of redundant rules, we provide the user with an explanation of the existing conflict. In this explanation, we identify the conflicting rules, which joint action they share, and the conditions in the triggers that make them redundant. Returning to the initial example, we would provide the following information to the user: “Rule 1 and Rule 2 are redundant: Both rules trigger the action of set hvac_bedroom temperature to 20°C. The differences are in trigger values: temperature_bedroom_sensor is less than 18°C for Rule 1; temperature_bedroom_sensor is less than 15°C for Rule 2. Therefore, when rule Rule 1 is triggered, Rule 2 will also be triggered, making these rules redundant.”

5.2.6 Export graphs to GEXF format

After generating all the graphs and detecting the conflicts between the rules, we exported the created graphs to the Graph Exchange XML Format (GEXF). In this way, users will be able to consult the generated graphs in order to be able to visualize the relationships between the nodes and why specific conflicts exist. GEXF is a language that makes it simple to represent complex structures such as graphs and their accompanying data [19].

Algorithm 3: Algorithm to discover redundancies in graphs

```

for every graph in graphs list do
  vertices  $\leftarrow$  graph.getVertices;
  rules_found  $\leftarrow$  [];
  for every vertex in vertices do
    edges  $\leftarrow$  vertex.getEdges;
    for  $i \leftarrow 0$  to edges.length - 1 do
      rule1  $\leftarrow$  edges[i].getAssociatedRule;
      for  $j \leftarrow i + 1$  to edges.length do
        rule2  $\leftarrow$  edges[j].getAssociatedRule;
        res  $\leftarrow$  haveOverlapTriggers(rule1, rule2);
        if res then
          same_action  $\leftarrow$  rule1.action = rule2.action;
          same_periodicity  $\leftarrow$ 
            checkPeriodicityIntersection(rule1.periodicity, rule2.periodicity);
          new_rule  $\leftarrow$  not rules_found contains pair (rule1, rule2);
          if same_action & same_periodicity & new_rule then
            append pair (rule1, rule2) to rules_found;
          end
        end
      end
    end
  end
end
  
```

In order to export each of the graphs to the GEXF format, we built a script to convert the information contained in the generated graphs, such as nodes, edges, and the name of the rule associated with each edge.

Once we obtain the graph in GEXF format, we can explore this graph using tools such as Gephi².

5.3 Known Limitations

While implementing our system and conducting tests, we noticed certain limitations that require attention.

The first limitation we can announce is related to the complexity of actions the system accepts. The actions that the system can parse are very simple and can only contain one command. For example, turning on a light and simultaneously setting its brightness is impossible. According to our implementation, the system only accepts the second part of this action (setting the brightness), which means that this rule applies to scenarios where the lights are already turned on with another percentage of brightness and to scenarios in which they are off. Multiple actions on the same rule on different devices (*e.g.*, turn on living room lights and TV) are also impossible. Actions related to sending notifications to users in case an event is detected are also unavailable in our system.

Our system's second limitation is related to the number of generated graphs. If sensors or actuators are defined in the YAML file that are later not used in any rule, these devices are still associated with a graph. This graph only contains one or two devices, depending on whether there are related devices. Although reduced resources are used with this limitation, these graphs have no use in the system, so that we can delete them.

Another area that our system could improve is the types of accepted values associated with each device. In this implementation, we only accept *boolean* or *numeric* values, which is quite limiting. However, some devices, such as weather sensors, lights, or kitchen equipment, can contain other values and have many associated configurable parameters. As we have already noted, we chose to build a simple system to facilitate the process of validating the hypothesis while allowing for some variation of rules/devices (*cf.* Section 4.5, p. 37). Thus, we opted only for these two value types for the devices, just like the aforementioned simple actions.

The last limitation identified is associated with the conflicts that we were able to detect. We only implemented the detection of loops, redundancies, and inconsistencies between rules that have similar triggers but different actions in the same actuator. However, other kinds of conflicts exist. As we mentioned, there is yet another type of inconsistency — between rules with different triggers and actions acting on the same actuator. Our system cannot detect this type of conflict.

These are the limitations we could identify as we implemented our system. Despite their existence, they do not prevent the system's normal functioning. They only influence the complexity and usability that the system presents.

²Gephi, <https://gephi.org/>

5.4 Summary

This chapter describes our system's implementation and discusses some of our choices while developing our conflict detection system (*cf.* Section 5.1, p. 39).

Using a graph-based approach, we built a system capable of representing all possible scenarios of state combinations of existing devices and the transitions between these scenarios. To this end, and to increase the efficiency of our system by reducing the number of existing scenarios, we start by obtaining groups of strongly connected devices through the implementation of the connected components algorithm. Next, we construct the respective directed graph for each of these groups, where each node will correspond to one of the possible scenarios for that group of devices. The transitions between these scenarios occur through the edges representing the rules defined for this smart environment.

Based on the implementation of graphs, detecting existing conflicts between rules becomes relatively simple. In the case of loops, it is enough to resort to a DFS algorithm to discover loops in graphs with a path containing more than three nodes. To detect the intended inconsistencies, we must analyze pairs of edges for each node that actuate in the same device and lead to different final nodes. To the pairs detected here, it is still necessary to add pairs of rules of this type that form cycles in the graphs. Finally, to detect redundancies, we need to examine, for each vertex, pairs of edges that go to the same final node, that is, pairs of rules with overlapping triggers and the same action on the same device. For all the conflicting situations detected in the graphs, we always have to check the periodicity of the involved rules to verify if they will be executed simultaneously on some occasions (*cf.* Section 5.2, p. 40).

Finally, in Section 5.3 (p. 58), we discuss the most relevant limitations in our implementation, indicating how each affects our system.

Chapter 6

Evaluation and Validation

6.1	Experimental Setup	60
6.1.1	System	61
6.1.2	Carried Out Experiences	61
6.2	Experiments	61
6.2.1	Rule Conflict Detection in a Designed Smart Environment	61
6.2.2	Extensive Analysis of the Detected Conflicts	87
6.3	Effectiveness of the Implementation of Connected Components Algorithm	94
6.4	Conclusions	96
6.5	Validity Threats	96
6.6	Summary	97

This chapter presents and discusses the experimental approach we designed to evaluate and validate our work. Section 6.1 presents a brief overview of the experiments that we perform and the purpose of each one. In Section 6.2 (p. 61), we explore each experiment in more detail and present/analyze the results obtained. In Section 6.3 (p. 94), we check whether our system with the connected components algorithm represents an effective improvement in terms of efficiency compared to the same system without this algorithm implemented. Section 6.4 (p. 96) summarizes the conclusions obtained from the experiments and analyses. Then, we explore the validation threats found in Section 6.5 (p. 96). Finally, in Section 6.6 (p. 97), we summarize the relevant information obtained throughout the chapter.

6.1 Experimental Setup

To validate our hypothesis (*cf.* Section 4.3, p. 35) and assess the extent to which we achieved the initially outlined goals (*cf.* Section 4.2, p. 33), we devised an assortment of experiments to put our method to the test.

6.1.1 System

The experiments carried out in our conflict detector were conducted on a computer with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz and 8GB of RAM, running Windows 11 Home.

6.1.2 Carried Out Experiences

We developed a three-step process to evaluate the ability of our solution to detect the types of conflicts we previously defined in Section 4.2 (p. 33) and answer the questions made in Section 4.4 (p. 35).

In the first step, we started by building, for a scenario that we had idealized, an initial set of trigger-action rules that did not contain conflicts between them. Then, we generated three more reduced sets of conflicting rules with the initial collection: (i) redundant rule set with the initial set; (ii) set of rules inconsistent with the initial set (more concretely, rules with the same trigger but a different action on the same actuator as some existing rule in the initial set); (iii) set of rules that form loops with the initially created rules. With the generated rules, we checked which conflicts exist between the generated rules and which ones our system identifies. In case there were differences between these two sets, we discussed what their causes might be.

In the second step, we developed a more extensive process to obtain all possible combinations for the rules taking into account the factors mentioned in **D2** and **D3** of Section 4.2 (p. 33). Based on these rules, we analyzed the ability of our system to detect the types of conflicts we initially proposed. Our objective with this step was to obtain a broader and more detailed view of our system's ability to detect conflicts, which would become quite complex to cover with just the first step.

Following these experiments, we effectively tested whether our system became more efficient after implementing the connected components algorithm, allowing us to reduce the number of possible scenarios and obtain results more quickly. To do so, we will compare the performance of our system with and without this implemented algorithm and compare the results obtained.

With these experiments, we intend to obtain results that will allow us to appraise our system efficiently and identify its strengths and weaknesses. We also hope that these results will help us to validate our hypothesis as well as to know which goals were met and which need to be improved.

6.2 Experiments

As mentioned, we performed two separate experiments to evaluate our system and validate our hypothesis. Let us now explore these experiments further and analyze the results.

6.2.1 Rule Conflict Detection in a Designed Smart Environment

This experiment aimed to analyze the behavior of our system in an intelligent environment designed by us that resembles a real environment. We intended to verify if the system would be able to parse a set of rules that we could use in our day-to-day life in a smart home, build the

graphs resulting from this parse and detect sets of conflicting rules that fit one of the three types of conflicts that we challenged ourselves to detect: (i) Loop conflicts; (ii) Conflicts between rules with (partially) the same trigger but different actions in the same device; (iii) Conflicts caused by redundant rules.

The simulated environment is a smart home with one garage, one patio, one pool, one garden, one living room, one kitchen, one bedroom, and one bathroom. We associate each division with a set of detectors/sensors and actuators that will be used to build the smart rules. The complete list of existing devices, the states in which they can be found or the range in which they work, can be consulted on Appendix B (p. 109).

6.2.1.1 Generate Initial Rule Set

To carry out this experiment, we needed an initial rule set of previously existing rules or to build a new one. This ruleset should contain rules of different types, *i.e.*, rules with one or more triggers and rules without triggers. The rules should also have different periodicities defined to test different types of these intersections, and there should be conflicting rules to test our system easily. We should ensure that the environment where these rules are inserted also has several sensors and actuators available to explore the effectiveness of implementing the connected components algorithm. This algorithm will help distribute the devices into several graphs and reduce the number of nodes. Although several datasets of smart rules for a smart environment are available, we could not find any that fulfilled our intentions. Some papers, such as this by Danny *et al.* [38], provide datasets with different types of rules using different devices but do not contain conflicts or a minimal number of them. Others contain very complex rules, like one collected from an older person who lived in a single apartment [40].

On the other hand, creating a new dataset by hand is a very complex task. It becomes complicated for humans to build a vast dataset of rules that contains different types of rules using different combinations of sensors, actuators, and periodicities. As we intend to have some diversity of rules to cover as many different cases as possible to evaluate how the system behaves, we needed another more viable solution.

To generate the initial rules we used in this experience — rules without conflicts—, we resort to ChatGPT¹. ChatGPT is an online AI chatbot that answers questions and satisfies user requests. We chose to resort to a chatbot instead of building the rules by hand because it can quickly generate a varied set of rules that cover our needs. It becomes simpler to achieve our goal using this method than adapting a previously existing dataset or building one by hand.

We described the smart environment for ChatGPT to satisfy our expectations and generate the rules in the format we wanted (IFTTT). We provided the list of existing devices presented above and a set of considerations for rule generation. We also provided some examples of rules to illustrate what we intended easily. As we made requests to improve the results obtained, we refined

¹ChatGPT, <https://openai.com/blog/chatgpt>

this query. The final query to generate the rule set used in this experience can be consulted in Appendix C (p. 113). The rules resulting from using this query can be found in Appendix D (p. 119).

After obtaining the initial rules, we built the respective YAML file and ran our system to verify the absence of conflicts, as requested. However, the results obtained showed the existence of 3 conflicts of the same trigger but different actions in the same device type: (i) between Rules 7 and 14; (ii) between Rules 7 and 35; (iii) between Rules 14 and 32.

Rule 7: If the bedroom temperature falls below 18°C, set the bedroom HVAC to 20°C. (Always)

Rule 14: Turn off the bedroom HVAC system every day at 11 PM. (Daily at 11 PM)

Rule 32: If the bedroom temperature exceeds 25°C, set the bedroom HVAC to 22°C. (Always)

Rule 35: f the bedroom temperature falls below 10°C, set the bedroom HVAC to 23°C. (Daily from 0 AM to 7 AM)

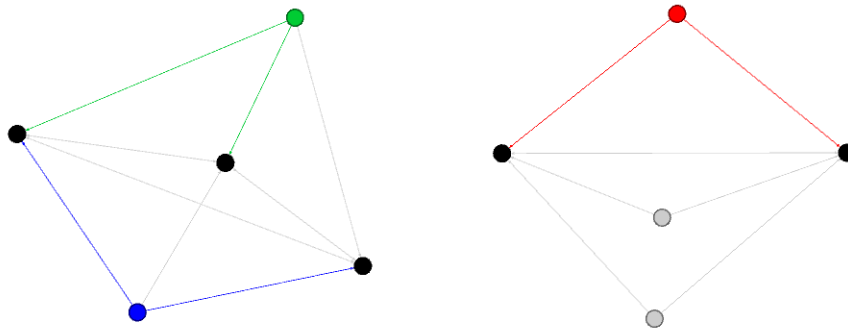


Figure 6.1: Graph where rules 7, 14, 32, and 35 are inserted

Figure 6.1 shows the graphs where the conflicting rules are inserted. Blue represents one of the edges corresponding to rule 7 and another of rule 14. In green is represented one of the edges associated with rule 7 and another with rule 35, and in red, the edges of rules 14 and 32. The initial vertex that demonstrates the conflict has the same color as the respective pair of rules. As we can see, each pair of rules we highlighted with the same color outgo from the same vertex but reach different final vertices. This different destination vertex indicates that these rules may be conflicting because there are two possible final scenarios if they can be executed simultaneously. Nevertheless, let us analyze the rules to see if the rules conflict.

Moving on to rule analysis, rules 7 and 14 can execute simultaneously — the periodicities intersect (Always vs. Daily at 11 PM) — in case the room temperature is below 18°C. Both act

on the same actuator (bedroom HVAC), but with different actions (turn off vs. set bedroom HVAC to 20°C). Checking the graph (*cf.* Figure 6.1, p. 63), we see that starting from the node marked in blue, with the execution of rules 7 and 14 simultaneously, we reach different final states. That is impossible to happen — we cannot have the HVAC equipment turned off and at 20°C. Therefore, these rules conflict.

Analyzing now rules 7 and 35, they act on the same actuator (bedroom HVAC) and will be executed at the same time daily between 0 AM and 7 AM in case the room temperature is below 10°C. The triggering of the two rules overlaps in this scenario because if the temperature is below 10°C, it is also below 18°C. Since the actions of the two rules place the room's HVAC at different temperatures simultaneously, we have a conflict. Analyzing Figure 6.1 (p. 63), we verify that for the state marked in green, there are two possibilities for final states.

Finally, the situation between rules 14 and 32 is similar to that between rules 7 and 14. Looking at the vertex highlighted in red, we can reach two different end states simultaneously — one where the room's HVAC is off and the other on which HVAC is set to a temperature of 22°C — in case the room temperature is higher than 25°C. Since this is an impossible situation, the rules conflict.

One way to correct this problem was to change the periodicities of the conflicting rules so they could not execute simultaneously. That was the suggestion we presented to ChatGPT for it to correct the rules, having obtained the rules found in Appendix E (p. 121). This new set no longer has conflicts, changing the frequency of rules 7 and 32 from 'Always' to 'from 4 PM to 10:59 PM'.

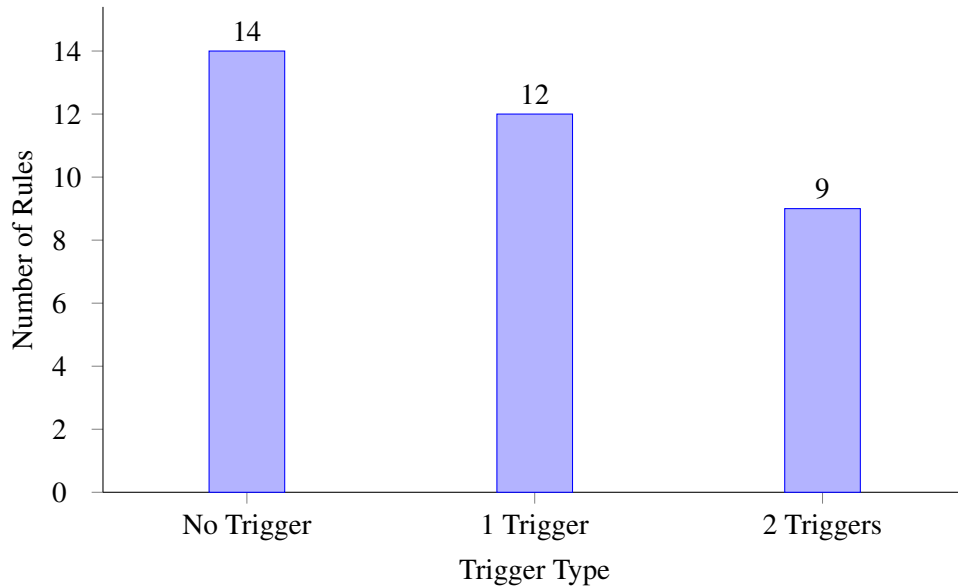


Figure 6.2: Number of rules per trigger type

To analyze the generated rules, we can relate them to the type of trigger each one has. To do so, we separate the rules into three different groups: (i) rules with no trigger part; (ii) rules with one trigger; (iii) rules with two triggers. The result obtained can be found in Figure 6.2. Analyzing

the graph obtained, we can verify that the number of rules with each trigger type is balanced, with not too many rules with a particular type of trigger and few with another type.

Figure 6.3 shows another possible analysis, consisting of grouping the rules according to the type of periodicity each one has: (i) Always; (ii) Time Interval (*e.g.*, Daily between 10 AM and 10 PM); (iii) Specific Time (*e.g.*, Every Tuesday at 7 AM). In this case, the distribution of rules for each type of periodicity is not well-balanced among the three types. While the number of rules with “Always” and “Specific Time” periodicity is similar, the number of rules with “Time Interval” periodicity is significantly lower. This discrepancy may indicate that, depending on the generated conflicting rules, this periodicity type may undergo testing in fewer combinations than the other types.

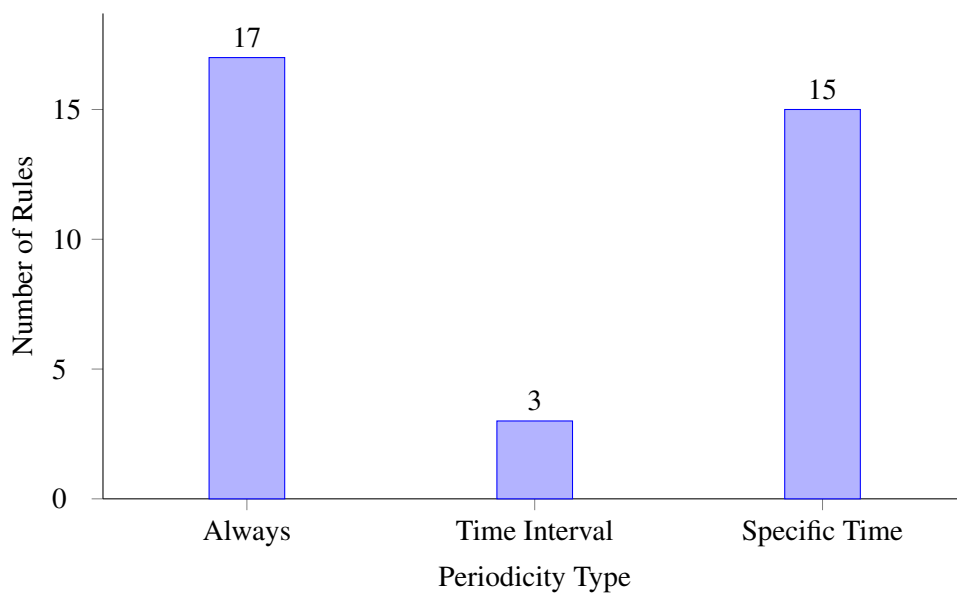


Figure 6.3: Number of rules per periodicity type

6.2.1.2 Generate a Set of Redundant Rules

With the initial rule set generated, we requested ChatGPT to generate a set that would be redundant with the previously generated rules. To do so, we explain what a conflict caused by redundant rules consists of and provide a conflicting example and an explanation for such a conflict to demonstrate what we wanted. In this request, we also asked that for each generated rule, the ChatGPT indicated with which rules it was in conflict so that we could compare with the conflicts we detected. By consulting the Appendix F (p. 124), it is possible to check the set of redundant rules created by ChatGPT.

Table 6.1 (p. 66) compares the redundancies detected by ChatGPT and the redundancies detected by our system for each of the new rules created. As we can see, there are three situations in which our assessment is different from that made by ChatGPT: (i) Possible redundancy between

Rule	Redundancies Detected by ChatGPT	Redundancies Detected by Our System
36	6, 37	6, 37
37	36	6, 36
38	31	31
39	32, 40	32
40	7, 39	7
41	34, 43	34, 43
42	33, 44	33, 44
43	34, 41	34, 41
44	42	33, 42

Table 6.1: Redundancies detected by ChatGPT vs. Redundancies detected by our system

rules **6** and **37**; (ii) Possible redundancy between rules **39** and **40**; (iii) Possible redundancy between rules **33** and **44**. In all other situations, the detection made by our system is the same as that made by ChatGPT, which could mean that our system will be very effective in detecting this type of conflict. At least 10 of the 12 redundancies we identified are correct, meaning a hit rate of at least 83.3%. We will now analyze each of the discordant situations to verify which of the systems is correct, and, in the end, we will be able to draw some conclusions regarding the ability to detect redundant situations in our system.

Possible redundancy between rules **6** and **37**

Rule 6: If the living room temperature exceeds 28°C, set the living room HVAC to 22°C. (Always)

Rule 37: If the living room temperature exceeds 30°C, set the living room HVAC to 22°C. (Every Saturday and Sunday between June and September from 10 AM to 6 PM)

Starting by analyzing the devices used by both rules, we verify that both use the same sensor — living room temperature sensor — and the same actuator — living room HVAC. Furthermore, we confirm that the two rules have the same action — “set the living room HVAC to 22°C”. Inspecting the rule triggers, in the case of rule 6, we see that it executes if the room temperature is higher than 28°C. In the case of rule 37, it executes if the room temperature is higher than 30°C. Consulting Table 5.2 (p. 56), we see this is one of the cases here ($>$, $>$ and $value1 < value2$). Therefore, the triggers overlap each other. It only remains to analyze the periodicity of the rules in order to be able to conclude if it is effectively a conflict. As one of the rules (rule 6) has “Always” periodicity, it will be executed at any time as long as the trigger condition is verified. Focusing on the periodicity of rule 37, although this one is of type “Time Interval”, it will always intersect with the periodicity “Always”. Thus, we can conclude that on any Friday or Saturday between June and September from 10 AM to 6 PM, if the room temperature is above 30°C, rules 6 and 37 will be executed simultaneously.

Through this analysis, we can state that rules 6 and 37 are redundant. In this case, the detection made by our system was correct.

Possible redundancy between rules 39 and 40

Rule 39: If the bedroom temperature exceeds 22°C, set the bedroom HVAC to 20°C. (Daily from 3 PM to 10:59 PM)

Rule 40: If the bedroom temperature falls below 15°C, set the bedroom HVAC to 20°C. (Daily from 5 PM to 8:59 PM)

In this situation, the two rules use the same sensor — bedroom temperature sensor — and the same actuator — bedroom HVAC. Comparing the actions of the rules, we see that they are the same in both cases — “set the bedroom HVAC to 20°C”. Moving on to the analysis of the triggers, in the case of rule 39, we find that it will be executed if the room temperature exceeds 22°C. On the other hand, rule 40 will be triggered in case the room temperature is above 20°C. Checking Table 5.2 (p. 56), this is not one of the situations present there ($>$, $<$ and $value2 < value1$), so, in this case, the triggers do not overlap. Analyzing the definition of redundant rules in Section 5.2.5.3 (p. 55), as the triggers do not overlap, the rules do not have similar triggers and, therefore, will not execute in the same situations.

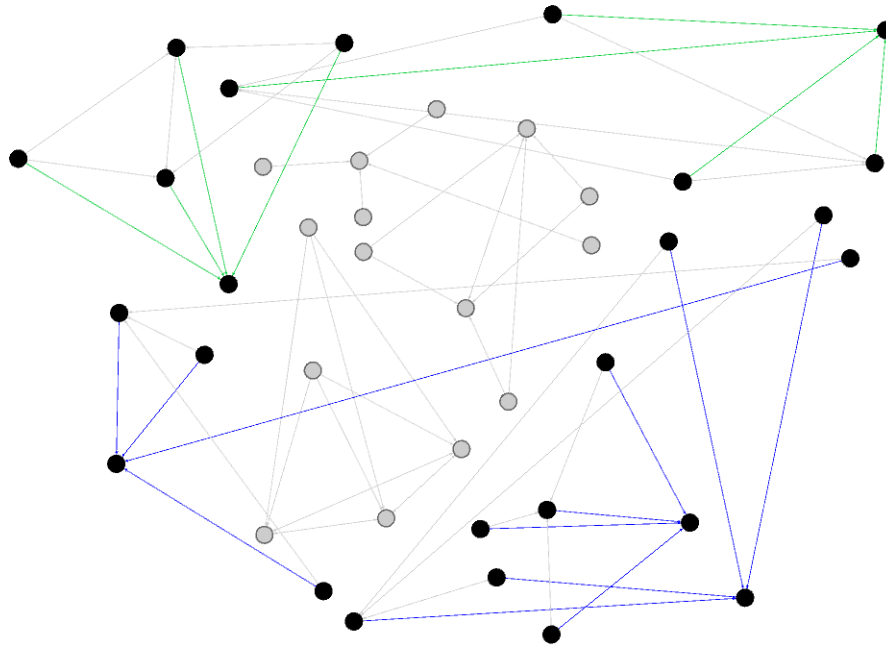


Figure 6.4: Graphs where rules 39 and 40 are inserted

To identify redundancy in a graph, we must have two different edges connecting the same initial and final vertices. When looking at the graphs in Figure 6.4, in which the edges associated

with rule 39 are in blue and those associated with rule 40 are in green, we confirm that there is no pair of edges from rules 39 and 40 that connect the same initial and final vertices.

In conclusion, rules 39 and 40 do not have overlapping triggers and are not executed in the same circumstances, so they cannot be redundant. Therefore, ChatGPT failed this detection.

Possible redundancy between rules 33 and 44

Rule 33: If the outside temperature is below 15°C and the pool water temperature is below 20°C, set the pool water heating system temperature to 25°C. (Always)

Rule 44: If the outside temperature is below 5°C and the pool water temperature is below 18°C, set the pool water heating system temperature to 25°C. (Every Friday, Saturday and Sunday in December and January from 10 AM to 7 PM)

Rules 33 and 44 are yet another case where the same sensors — outdoor temperature sensor and pool water temperature sensor — and the same actuator — pool water heating system. In this case, two triggers constitute each of the rules. Upon analyzing each one, we find that the conditions for rule 33 to be executed are the outside temperature below 15°C and the pool water temperature below 20°C. In the case of rule 44, the outside temperature must be below 5°C, and the pool water temperature must be below 18°C. We can find these two situations in Table 5.2 (p. 56) (outside temperature: $<, <$ and $value2 < value1$; pool water temperature: $<, <$ and $value2 < value1$), so we check the condition of the overlapping triggers. By comparing actions, we find that both rules have the same one — “set the pool water heating system temperature to 25°C” —, so it remains to analyze the periodicities of the rules to verify whether there is a redundancy. As the periodicity of rule 33 is “Always”, we can conclude that the two rules will execute simultaneously every Friday, Saturday, and Sunday of December and January between 10 AM and 7 PM if the conditions previously analyzed are met.

As rules 33 and 44 have overlapping triggers, the same action, and their periodicities intersect at some point, we conclude that they are redundant. The detection made by our system was correct.

After completing the analysis of the divergent cases between the conflicts detected by ChatGPT and our system, we must mark the success rate of our system. As we could verify, all the redundancies we detected were correct, so, in this case, the efficiency of our system was 100%. In addition, we were able to detect more redundancies than those detected by ChatGPT, which even detected a pair of redundant rules that, after analyzing it, we found to be invalid. These are good indicators that our system is correctly detecting existing redundancies.

The frequency of each possible combination for the Rule Triggers / Rules Periodicity pair of detected redundancies can be found in Table 6.2 (p. 69). As we can see, the redundancies found only fit in two of the combinations, so a substantial part of the existing possibilities was not explored. This situation may be due to some flaw in the explanation given to ChatGPT of redundancy between two rules and when two periodicities intersect. On the other hand, as ChatGPT responses

Rule Triggers / Rules Periodicity	Frequency
Rules Sharing the Same Sensors / Always vs. Always	3
Rules Sharing the Same Sensors / Always vs. Specific Time	0
Rules Sharing the Same Sensors / Always vs. Time Interval	9
Rules Sharing the Same Sensors / Specific Time vs. Specific Time (same)	0
Rules Sharing the Same Sensors / Specific Time vs. Time Interval	0
Rules Sharing the Same Sensors / Time Interval vs. Time Interval (same)	0
Rules Sharing the Same Sensors / Time Interval vs. Time Interval (partially intersect)	0
Rules Sharing the Same Sensors / Time Interval vs. Time Interval (one contains the other)	0
Both Rules Have No Trigger / Always vs. Always	0
Both Rules Have No Trigger / Always vs. Specific Time	0
Both Rules Have No Trigger / Always vs. Time Interval	0
Both Rules Have No Trigger / Specific Time vs. Specific Time (same)	0
Both Rules Have No Trigger / Specific Time vs. Time Interval	0
Both Rules Have No Trigger / Time Interval vs. Time Interval (same)	0
Both Rules Have No Trigger / Time Interval vs. Time Interval (partially intersect)	0
Both Rules Have No Trigger / Time Interval vs. Time Interval (one contains the other)	0

Table 6.2: Frequency of each pair Rule Triggers / Rules Periodicity in the founded redundant rules

are not always the same, perhaps the same request made again could have returned more varied responses. Asking for a greater number of rules could also help to cover more situations.

Thus, this experiment only allows us to conclude that we could identify redundancies for the pairs *Rules Sharing the Same Sensors / Always vs. Always* and *Rules Sharing the Same Sensors / Always vs. Time Interval*.

6.2.1.3 Generate a Set of Inconsistent Rules

After analyzing the redundancies, it is time to analyze the detection of inconsistencies by our system. As mentioned in Section 5.2.5.2 (p. 50), our system focuses only on inconsistencies between rules with the same triggers but different actions on the same actuator. Thus, we requested ChatGPT to generate a new set of rules that would cause this type of conflict with the initially generated rules. In this request, we define this type of conflict, informing which relationships between rule triggers may exist in this case — rules sharing the same sensors, rules sharing only part of the sensors, one of the rules has no trigger part, and both rules have no trigger part. We also requested to vary the periodicities to cover multiple situations and indicate the conflicting rules for each generated rule. The generated rules can be consulted in the Appendix G (p. 125).

In Table 6.3 (p. 70), a comparison is made between the inconsistencies detected by ChatGPT and the inconsistencies detected by our system regarding each of the new rules. Analyzing the table, we notice that there are some cases in which our judgment is different from that of ChatGPT: (i) Possible inconsistency between rules 1 and 45; (ii) Possible inconsistency between rules 9 and 47; (iii) Possible inconsistency between rules 26 and 47; (iv) Possible inconsistency between rules 21 and 48; (v) Possible inconsistency between rules 35 and 50; (vi) Possible inconsistency between rules 25 and 52; (vii) Possible inconsistency between rules 21 and 54; (viii) Possible inconsistency

Rule	Inconsistencies Detected by ChatGPT	Inconsistencies Detected by Our System
45	20, 46	1, 20, 46
46	1, 45	1, 45
47	4, 18, 51	4, 9, 18, 26, 51
48	5, 19, 54	5, 19, 21, 54
49	6, 56, 57	6, 56, 57
50	7, 35	7
51	4, 9, 18, 47, 58	4, 9, 18, 47, 58
52	10	10, 25
53	20	20
54	5, 19, 48	5, 19, 21, 48
55	22	22
56	6, 49, 57	6, 49, 57
57	6, 31, 49, 56	6, 49, 56
58	9, 18, 26, 51	9, 18, 26, 51

Table 6.3: Inconsistencies detected by ChatGPT vs. Inconsistencies detected by our system

between rules **31** and **57**. In all other situations, our decision was the same as ChatGPT had. Let us now discuss each divergent case to see which system decided correctly. Subsequently, based on the results obtained, we will draw some conclusions about the ability of our system to analyze inconsistencies.

Possible inconsistency between rules 1 and 45

Rule 1: If motion is detected in the living room and living room lights are off, set the living room lights brightness to 50%. (Always)

Rule 45: If motion is detected in the living room and living room lights are off, set the living room lights brightness to 70%. (Daily at 5 PM)

Rules 1 and 45 use the same sensors — living room motion sensor and living room lights — and the same actuator — living room lights. The rule triggers allow us to fit it into the *rules sharing the same sensors* type. Through the states defined for the sensors used in the triggers, we can conclude that the initial scenarios for these rules are the same — when there is movement in the living room, and the lights in this room are off. However, the action defined for each of the rules is different, using the same actuator — the brightness of the living room lights at 50% (rule 1) or 70% (rule 45) — which means there are different final scenarios for the same initial case. We verify this by examining in Figure 6.5 (p. 71) the graphs where these two rules are inserted. The edges associated with rule 1 (blue) and those associated with rule 45 (green) leave from the same initial vertex but have different destinations. If the periodicities of the two rules intersect at some point, this represents a conflict situation. Comparing the periodicities of the rules, rule 1 is “Always” while rule 45 is “Daily at 5 PM”. This periodicity intersection allows us to say that

as the rules have the same triggers, they will run simultaneously every day at 5 PM, causing a conflicting situation.

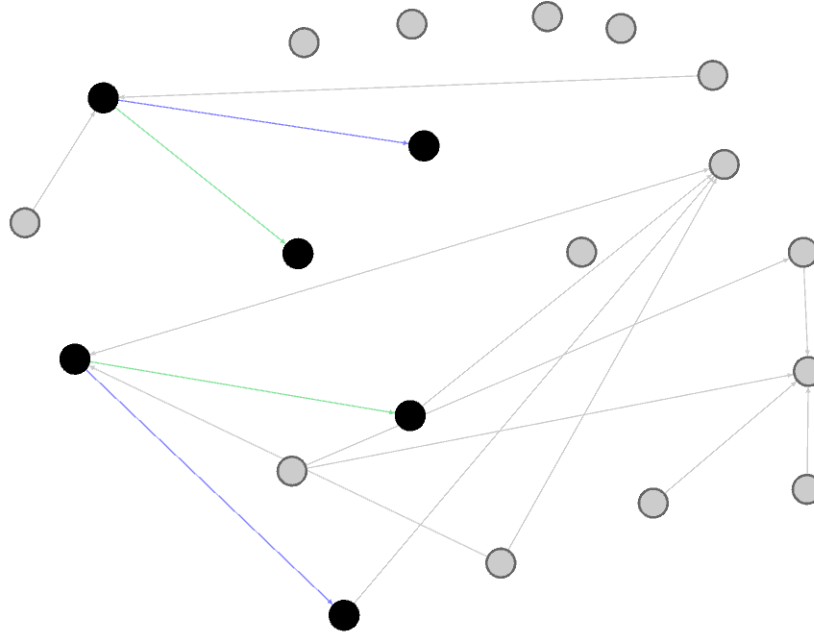


Figure 6.5: Graphs where rules 1 and 45 are inserted

Thus, we conclude that rules 1 and 45 are inconsistent. Therefore, the detection made by our system is correct.

Possible inconsistency between rules 9 and 47

Rule 9: Set the garage lights brightness to 35% every day at 8 PM. (Daily at 8 PM)

Rule 47: When the garage door is open, set the garage lights brightness to 80%. (Always)

Analyzing rules 9 and 47, we see that one of them has no trigger (rule 9) while the other has a trigger that uses the garage door sensor (rule 47). So, rule 47 will execute when the garage door is open, while the other one will trigger in any scenario, depending on the respective periodicities. Regarding each rule's action, both use the same actuator — garage lights — but with different actions — brightness of the garage lights at 35% (rule 9) or 80% (rule 47). If these rules are triggered simultaneously, there will be a conflict because they execute different commands in the same actuator, implying different final scenarios. This situation with multiple final scenarios is what we can verify in the graphs of Figure 6.6 (p. 72), noting that there are edges associated with rule 9 (blue) and rule 47 (green) that start from the same initial nodes (in red) but have different final nodes. Focusing on the periodicities of the two rules, we conclude that when the garage door is open at 8 PM, there will be a conflict between rules 9 and 47.

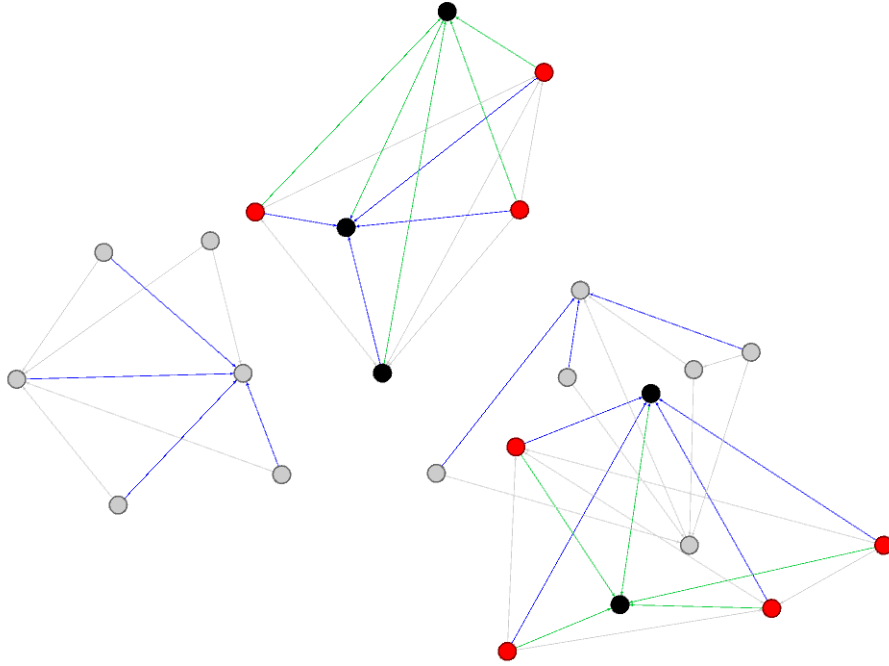


Figure 6.6: Graphs where rules 9 and 47 are inserted

Therefore, we can conclude that rules 9 and 47 are inconsistent and that our system worked correctly.

Possible inconsistency between rules 26 and 47

Rule 26: Set the garage lights brightness to 35% every Tuesday and Thursday at 7 PM.

Rule 47: When the garage door is open, set the garage lights brightness to 80%. (Always)

This situation is similar to the previous situation, changing only the periodicity of one of the rules. The sensor and actuator we use are the same, as well as the intended actions. When analyzing the graphs where the rules are inserted (*cf.* Figure 6.7, p. 73), we verify the same situation previously described: there are edges associated with rules 26 (blue) and 47 (green) that have the same initial vertexes (in red), but the ending vertex is different. If the periodicities of the rules intersect, we have a conflict. Checking the periodicities, we conclude that, being one “Always” and the other “Daily at 7 PM”, the rules will be executed simultaneously at 7 PM on all Tuesdays and Thursdays when the garage door is open.

In conclusion, rules 26 and 47 are inconsistent, so our system got it right once again.

Possible inconsistency between rules 21 and 48

Rule 21: Set the patio lights brightness to 75% every day at 9 PM. (Daily at 9 PM)

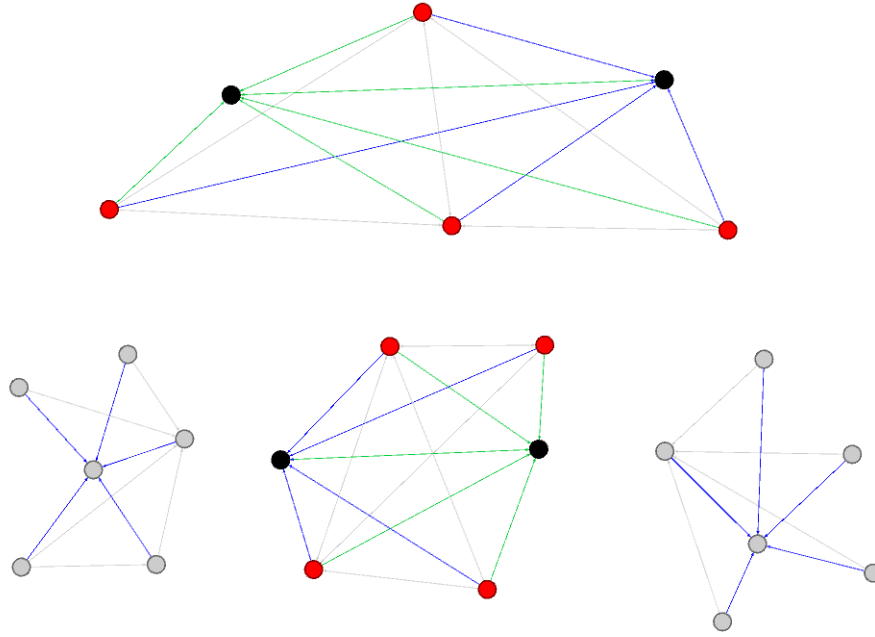


Figure 6.7: Graphs where rules 26 and 47 are inserted

Rule 48: If the patio entry door is open and patio lights are off, set the patio lights brightness to 30%. (Daily from 6 PM to 6 AM)

Rules 21 and 48 refer to a situation where *one of the rules has no trigger part*. While rule 48 uses two sensors in its triggers — patio entry door sensor and patio lights sensor —, rule 21 has no triggers, meaning it executes at any time, according to the respective periodicity. Both rules use patio lights as an actuator. However, they set different actions — brightness to 75% (rule 21) or 30% (rule 48). Checking the graphs in which these rules are inserted (*cf.* Figure 6.8, p. 74), we notice that there is a situation in which an edge corresponding to rule 21 (blue) and one corresponding to rule 48 (green) leave the same node (in red), heading to different final nodes. For this reason, if their periodicities intersect at some point, there will be a conflict. Paying attention to the respective periodicities, it is the intersection of a *specific time* (Daily at 9 PM) with a *time interval* (Daily from 6 PM to 6 AM). Thus, 9 PM is contained in the range 6 PM - 6 AM, so the rules will run concurrently on days when at 9 PM, the patio door is open, and the patio lights are off.

Considering this justification, we conclude that rules 48 and 21 are inconsistent, as our system had detected.

Possible inconsistency between rules 35 and 50

Rule 35: If the bedroom temperature falls below 10°C, set the bedroom HVAC to 23°C. (Daily from 0 AM to 7 AM)

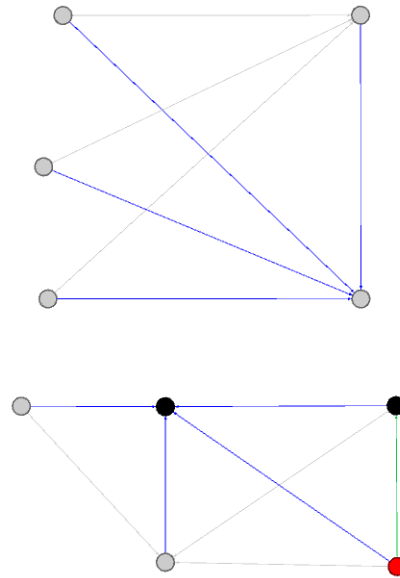


Figure 6.8: Graphs where rules 21 and 48 are inserted

Rule 50: If the bedroom temperature is below or equal to 18°C , set the bedroom HVAC to 23°C .
(Daily from 4 PM to 10:59 PM)

Rules 35 and 50 use the same sensor — bedroom temperature sensor — and the same actuator — bedroom HVAC. If we compare the triggers of the rules, we see that, depending on the respective periodicity, they are triggered simultaneously when the room temperature is below 10°C . That triggers comparison is one of the combinations of symbols and values in Table 5.1 (p. 52) ($<$, $\leq$ and $value1 < value2$). Analyzing the actions of the rules, we find out that they define the same behavior for the same actuator — set the bedroom HVAC to 23°C . However, the same action on the same actuator corresponds to a redundancy, not to an inconsistency. So, while it is true that there is a conflict between rules 35 and 50, this is not the type we are analyzing here.

Thus, rules 35 and 50 are not inconsistent. ChatGPT made this detection wrongly.

Possible inconsistency between rules 25 and 52

Rule 25: Turn on the garden watering system every Monday, Wednesday, and Friday at 6 AM.
(Every Monday, Wednesday and Friday at 6 AM)

Rule 52: If the humidity in the garden is below 30%, turn off the garden watering system. (Always)

Rules 25 and 52 are another example where *one of the rules has no trigger part*. The rule containing triggers uses a sensor — garden humidity sensor. Both rules use the same actuator

— garden watering system —, and the actions defined for it are contradictory. While rule 25 states that the garden watering system must turn on, rule 52 states that this system must turn off. Therefore, if the periodicities of the rules intersect, we are facing a conflict between two rules with the same trigger but different actions on the same actuator. This multitude of final states is also what the Figure 6.9 graphs show us, where the relationship between the two rules this time forms a loop. This loop happens because when the humidity is below 30%, there are only two possible scenarios: (i) garden watering system is off; (ii) garden watering system is on. Rule 25 represents a transition from scenario 1 to scenario 2 (blue arrow on the graph), while rule 52 represents a transition from scenario 2 to scenario 1 (green arrow on the graph). Thus, if the rules execute simultaneously, there are two possible final vertices — conflict situation. Analyzing the periodicities of the rules, we find that they intersect and will be executed simultaneously every Monday, Wednesday, and Friday at 6 AM in case the humidity in the garden is below 30%.

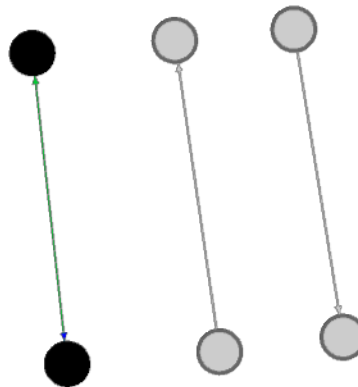


Figure 6.9: Graphs where rules 25 and 52 are inserted

Therefore, as our system detected, rules 25 and 52 are inconsistent.

Possible inconsistency between rules 21 and 54

Rule 21: Set the patio lights brightness to 75% every day at 9 PM. (Daily at 9 PM)

Rule 54: Set the patio lights brightness to 90% every day at 9 PM. (Daily at 9 PM)

None of the rules 21 or 54 have a trigger part. Thus, it is enough to assess whether their actions act on the same actuator and whether these actions are contradictory to know whether they are inconsistent or not. As we have seen, both rules use the patio lights as an actuator, and their actions are different — set the patio lights brightness to 75% or 90%. The graphs to which these rules belong (*cf.* Figure 6.10, p. 76) show us that there are some situations where there are two possible final scenarios when the rules execute at the same time from the same initial scenario (in red). These multiple-final scenarios are impossible, so it constitutes a conflict if the rules can

execute simultaneously. These rules have the same periodicity, so they will be executed daily at 9 PM, causing a conflict.

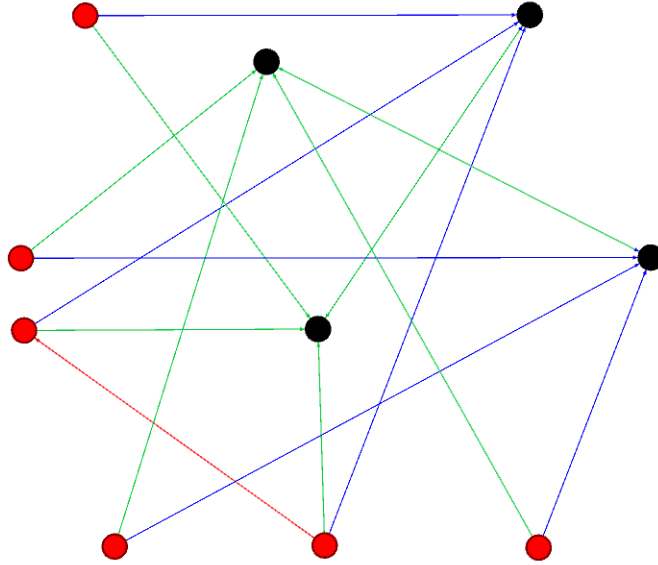


Figure 6.10: Graphs where rules 21 and 54 are inserted

For all this, we conclude that rules 21 and 54 are inconsistent, as we detected with our system.

Possible inconsistency between rules 31 and 57

Rule 31: If the living room temperature falls below 15°C, set the living room HVAC to 23°C. (Always)

Rule 57: If the living room temperature exceeds 25°C, set the living room HVAC to 21°C. (Daily between June and September from 9 AM to 7 PM)

Rules 31 and 57 use the same sensor — living room temperature sensor — and the same actuator — living room HVAC. However, analyzing both triggers, we verify that they will always be executed at different times. It is impossible to have a scenario where the temperature in the living room is simultaneously below 15°C and above 25°C. We can conclude this even by analyzing the graph where these rules are inserted (*cf.* Figure 6.11, p. 77). This graph has no vertex from which two edges of different colors emerge (the blue edges associated with rule 31 and the green ones associated with rule 57). Therefore, there can never be any conflict between these two rules.

Thus, we conclude that rules 31 and 57, contrary to what was detected by ChatGPT, are not inconsistent.

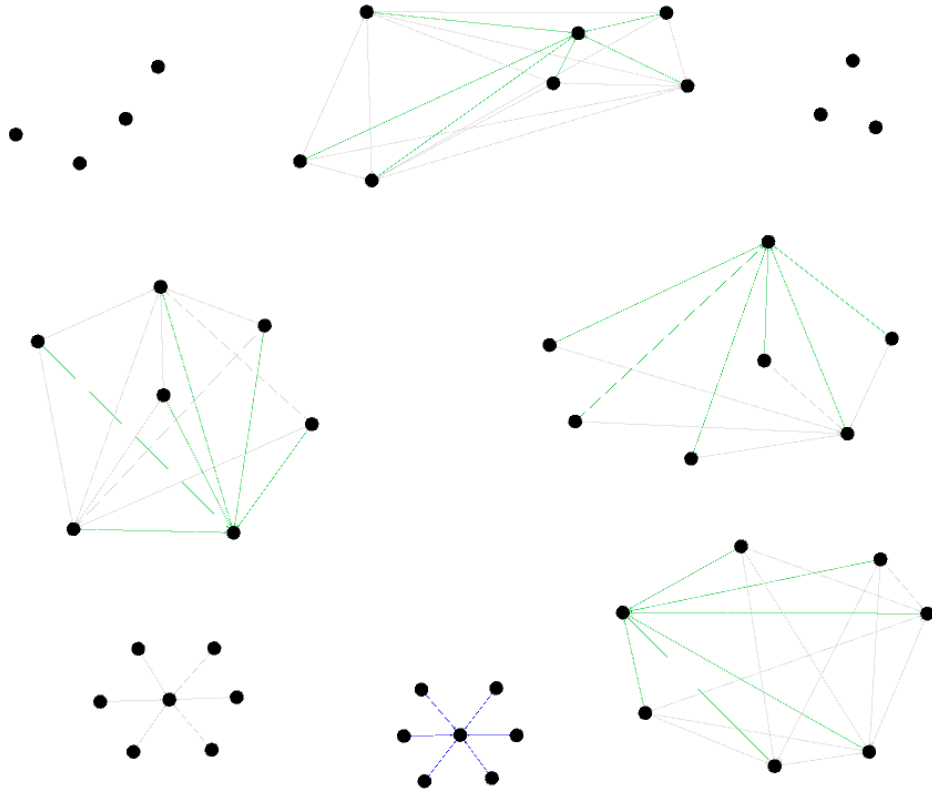


Figure 6.11: Graph where rules 31 and 57 are inserted

Having completed the analysis of the divergent cases between the inconsistencies detected by our system and the divergences detected by ChatGPT, it is essential to highlight the effectiveness with which our system detected this type of conflict. Out of the 34 existing inconsistencies, our system detected all situations. Furthermore, all conflicting situations detected by our system represented inconsistencies. Thus, we can conclude that the hit rate was 100%. It is also noteworthy that our system was able to identify more inconsistencies than ChatGPT and that some of the combinations detected by ChatGPT did not represent inconsistencies or even conflicts.

Throughout this analysis, we could also detect conflicts caused by rules with the same trigger but different actions on the same device, both caused by loops in the graph and by edges that start from the same node and have different final nodes.

However, analyzing Table 6.4 (p. 78), where we analyze the frequency of each possible combination for the Rule Triggers / Rules Periodicity pair of detected inconsistencies, we cannot cover all possible situations. From 32 possible combinations (4 rule triggers possibilities * 8 rules periodicity possibilities), the detected inconsistencies fit into 9 combinations (28%). We could probably cover a more significant number of cases if the ruleset was bigger. Despite this, we found that with the rules generated, we could detect inconsistencies for all types of rule triggers and 5 of the 8 types available for the periodicity of the rules. This ability to cover several cases, albeit separately, is already a great sign, as it means that both the function to detect if two triggers are

Rule Triggers / Rules Periodicity	Frequency
Rules Sharing the Same Sensors / Always vs. Always	6
Rules Sharing the Same Sensors / Always vs. Specific Time	2
Rules Sharing the Same Sensors / Always vs. Time Interval	3
Rules Sharing the Same Sensors / Time Interval vs. Time Interval (same)	1
Rules Sharing Only Part of the Sensors / Always vs. Always	2
Rules Sharing Only Part of the Sensors / Always vs. Time Interval	5
One Rule Has No Trigger Part / Always vs. Specific Time	8
One Rule Has No Trigger Part / Specific Time vs. Time Interval	5
Both Rules Have No Trigger / Specific Time vs. Specific Time (same)	1
Other 23 Combinations	0

Table 6.4: Frequency of each pair Rule Triggers / Rules Periodicity in the founded inconsistent rules

triggered in the same circumstances, as well as the function to detect different actions on the same actuator and the one that checks if two intersecting periodicities are working correctly.

6.2.1.4 Generate a Set of Loops

The last type of conflict, whose detection we will analyze in this experiment, is the loops. As mentioned in Section 5.2.5.1 (p. 47), our solution considers that to be a loop, it must contain at least three conflicting rules. To do so, we again asked ChatGPT to generate sets of rules with loops this time. These sets could involve the initially generated rules or else new rules. As in the other times, we explained what we consider a loop accompanied by an example to make it easily understandable. The generated rule sets can be found in Appendix H (p. 127).

We will now analyze the generated sets to verify if they effectively are loops and if our system detects them.

Loop Set 1

Rule 59: If motion is detected in the living room, set the living room lights brightness to 25%. (Always)

Rule 60: If the living room lights are off and the living room door is open, set the living room lights brightness to 70%. (Always)

Rule 61: If motion is detected in the living room, turn off the living room lights. (Always)

Let us consider an initial scenario where the living room lights are off, and the living room door is open. In this case, rule 60 will trigger, and the living room lights will be turned on at 70% brightness. If, in this scenario, motion is detected, both rules 59 and 61 will be executed. However, these rules are conflicting because they have different actions — brightness to 25% or

turn off — on the same actuator — living room lights. Furthermore, these rules also conflict with rule 60, which sets a brightness of 70% for living room lights. Thus, considering that the living room door remains open and that movement continues to be detected in the living room, if rule 61 executes, rule 60 will also execute (action of rule 61 is a trigger of rule 60). At the same time, rule 59 is also being executed, existing three possibilities for the status of the lights — brightness to 25%, brightness to 70% or off. In this scenario, these rules will be continuously executed as the periodicity of the three rules is “Always”, so we are in the presence of a loop.

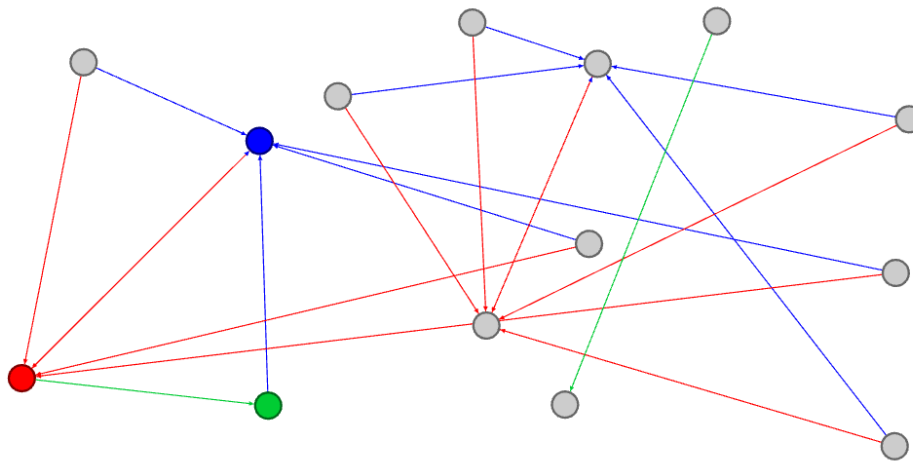


Figure 6.12: Graph where rules 59, 60, and 61 are inserted

As seen in Figure 6.12, in the graph generated by our system where rules 59, 60, and 61 are included, it is possible to identify this loop. An edge associated with rule 61 (red edges) leaves towards the vertex marked in red from the vertex marked in blue. It is possible to reach the vertex marked in green through an edge associated with rule 60 (green edges) from this vertex. Then, we return to the initial blue vertex using an edge associated with rule 59 (blue edges).

Thus, we conclude that this set contains rules that form a loop and that our system can detect it.

Loop Set 2

Rule 4: When the garage door is open, set the garage lights brightness to 35%. (Always)

Rule 62: If the garage lights are off and the garage door is open, set the garage lights brightness to 80%. (Always)

Rule 63: When the garage door is open, turn off the garage lights. (Always)

Consider a starting scenario where the garage door is open, and the garage lights are off. In this scenario, all rules will execute. Rules 4 and 63 have the same trigger — garage door is open —, while rule 62 adds one more — garage lights are off. The three rules use the same actuator — garage lights — but define different actions for it — brightness to 50%, brightness to 25%, or

turn off. Keeping the garage door open, as the periodicity of the three rules is “Always”, these rules will be continuously executed, constantly changing the garage lights’ state without reaching a definitive final state.

We can see two loops associated with these three rules by analyzing the graph where these rules are inserted (cf. Figure 6.13). Starting from the nodes in blue, two edges associated with rule 63 (red edges) allow reaching the red nodes. From these, it is possible to reach the green nodes using two edges associated with rule 62 (green edges). From the green nodes, it is possible to return to the initial blue nodes using two edges related to rule 4 (blue edges).

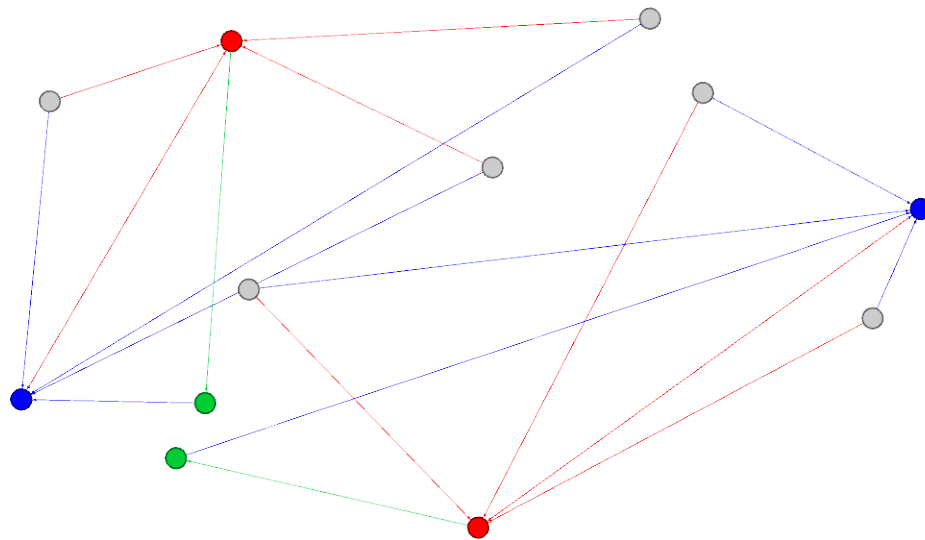


Figure 6.13: Graph where rules 4, 62, and 63 are inserted

Considering this analysis, our system can detect loops between rules 4, 62, and 63.

Loop Set 3

Rule 1: If motion is detected in the living room and living room lights are off, set the living room lights brightness to 50%. (Always)

Rule 59: If motion is detected in the living room, set the living room lights brightness to 25%. (Always)

Rule 61: If motion is detected in the living room, turn off the living room lights. (Always)

Consider the following scenario: motion is detected in the living room, and the living room lights are off. Every rule will execute in this case. The triggers for rules 59 and 61 are the same —

motion is detected in the living room —, while rule 62 adds an extra event — living room lights off. The three rules rely on the same actuator — living room lights — but describe various actions for it — brightness to 50%, brightness to 25%, or turning off. As long as motion is detected in the living room and the periodicity of the three rules is “Always”, these rules will be triggered indefinitely, modifying the state of the living room lights without reaching an end state.

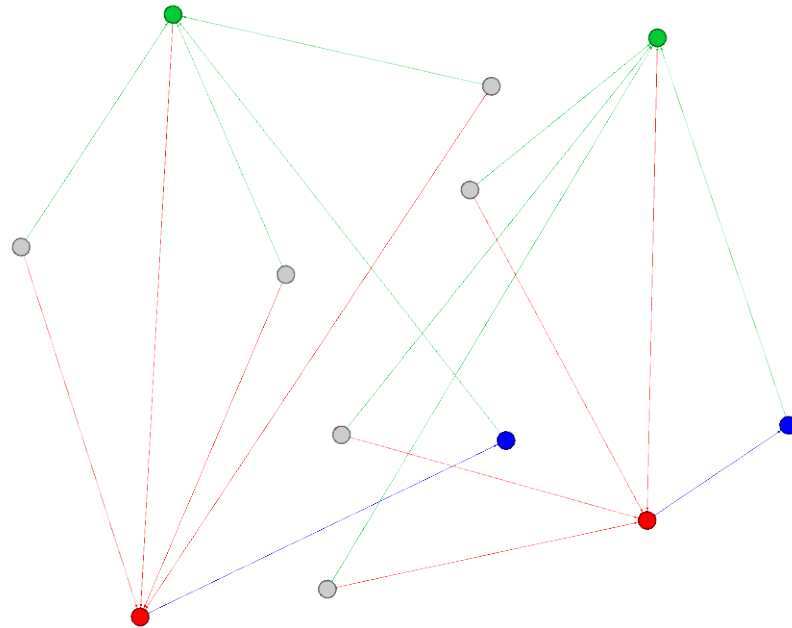


Figure 6.14: Graph where rules 1, 59, and 61 are inserted

It is possible to visualize this continuous execution of the rules without reaching a concrete final state in Figure 6.14. In this graph, it is possible to verify that there are two situations in which loops occur between these rules. Starting from the blue vertices and executing the rules according to rule 59 (green edges) - rule 61 (red edges) - rule 1 (blue edges) sequence, it is possible to return to these vertices.

Based on this analysis, we conclude that the execution of these rules creates a cycle detected by our system in which the lights in the living room keep changing status forever.

Loop Set 4

Rule 64: If the living room lights are turned on, close the living room blinds. (Daily at 8 PM)

Rule 65: If the living room blinds are closed, set the living room HVAC to 22°C. (Daily between December and February from 6 PM to 12 PM)

Rule 66: If the living room HVAC is turned on, set the living room lights brightness to 50%. (Always)

When analyzing rules 64, 65, and 66, we verified that they could be executed in sequence since the action of rule 64 — close the living room curtains — is the trigger of rule 65, the action of rule 65 — set the living room HVAC to 22°C — is the trigger of rule 66 and the action of rule 66 — set the living room lights brightness to 50% — is the trigger of rule 64. However, this may not be enough for a loop to exist and for the rules to run continuously.

If we analyze the rules, rule 64 executes whenever the living room lights are on, regardless of the set brightness. After the execution of rule 65, the lights in the living room are still on and with the same brightness as this rule only interferes with the state of the HVAC of this room. At this point, after rule 65 has run, rule 66 may or may not run. If the brightness of the lights in the living room at that time is equal to 50%, then the rule will not be executed (in our system) because the initial state of the lights is the same as its final state. Therefore there are no changes in the scenario. Other systems may execute it, but the result will be the same as the one obtained on ours — the state of the living room lights remains the same. In case the brightness of the lights in the living room is different from 50%, then rule 66 executes, changing the state of the lights. After executing the three rules, the final scenario will necessarily differ from what we initially had. Otherwise, the rules would not be executed (or if executed, there would be no change in the state of any device, so the initial and final scenarios would be the same). So we have a final scenario where the living room lights are on at 50% brightness, the HVAC is on at 22°C, and the blinds are closed. In this situation, the execution of any of the rules will not change the status of the devices, so there will be no simultaneous conflicting scenarios. In our opinion, this set of rules will execute only once, reaching a final stable scenario. Since the rules will not be continuously executed, there will not be an uninterrupted change in the states where the devices are involved.

We can prove what has been described here by examining the graph where these rules are inserted. In Figure 6.15 (p. 83), we see that there are situations in which not all the rules are executed (when the devices affected by the rules are already in the desired final state). There are others in which the three rules are executed in a row, but in none of these situations is the end node the same as the start node. This way, detecting any loops in this graph is impossible.

Given the analysis made first to the rules and then to the generated graph, we argue that rules 64, 65, and 66 do not constitute a valid example of a loop situation.

Analyzing the generated sets by ChatGPT, our system can detect loops involving three rules. However, we argue that our system can detect loops involving any number of rules equal to or greater than three and with different types of periodicity. To demonstrate this capability, we decided to join all generated rules, excluding only those belonging to this last loop set, as they do not add value to the dataset (*cf.* Appendix I, p. 129). Then, we asked ChatGPT to detect existing loops between all the rules and compared the results with those obtained in our system. The final result is displayed in Table 6.5 (p. 84).

Analyzing this table, we verified that we could detect loops involving three, four, or five rules. For larger sizes, with this dataset, it is impossible to say that we can detect. However, we expect this detection to happen since the algorithm used to discover loops with a higher number of rules

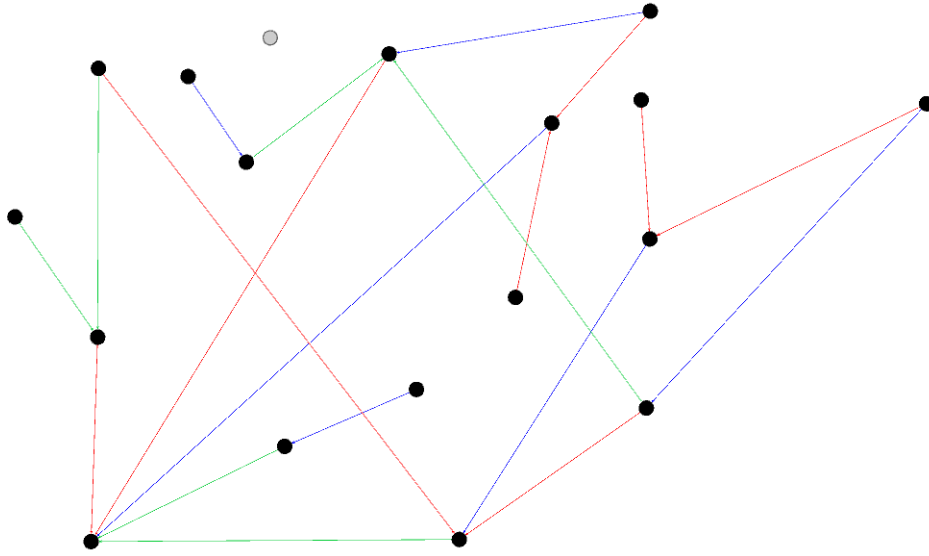


Figure 6.15: Graph where rules 64, 65, and 66 are inserted

is the same one that can discover these three sizes.

Another point worth noting is that, due to parallel edges, the number of rules containing loops between them is greater than that shown in this table. As mentioned in Section 5.2.5.1 (p. 47), the existence of parallel edges allows the same set of vertices to be associated with several combinations of rules. As is also mentioned in this section, in the presented results, we only return one of the possible combinations to try to simplify the task of identifying the loops by the users.

Despite this simplification in the presented results, it is still possible to identify the remaining possible combinations for the same set of nodes. To prove this, let us go back to *Loop Set 3* and the respective graph of Figure 6.14 (p. 81). If we notice, in Table 6.5 (p. 84), the existing loop between rules 1, 59, and 61 is not present. Even so, we can find the equivalent loop in the table, represented by rules 1, 46, and 59. These two sets of rules originate loops between the same vertices because rules 46 and 61 are redundant — have the same trigger and the same action, and the periodicities intersect. For this reason, the edges associated with each one connect the same pairs of vertices, as shown in Figure 6.16 (p. 85).

In this way, we demonstrated that it is still possible to detect the remaining sequences whose execution creates cycles, although we only present one combination.

Throughout this loop analysis, we demonstrated that our system could detect loops involving three or more rules, making it possible to identify all existing rule combinations for each set of nodes that form a loop.

Loop Size	Detected Loops
3	4 - 47 - 51
	4 - 47 - 63
	4 - 51 - 63
	47 - 51 - 63
	47 - 58 - 63
	4 - 58 - 63
	51 - 58 - 63
	49 - 57 - 6
	49 - 56 - 6
	56 - 57 - 6
	49 - 56 - 57
	1 - 46 - 53
	1 - 46 - 59
	45 - 46 - 53
	45 - 46 - 59
	46 - 53 - 59
	4 - 47 - 51 - 63
	4 - 47 - 58 - 63
	4 - 51 - 58 - 63
4	47 - 51 - 58 - 63
	49 - 56 - 57 - 6
	1 - 46 - 53 - 59
	45 - 46 - 53 - 59
5	4 - 47 - 51 - 58 - 63

Table 6.5: Loops detected between all rules in the set

6.2.1.5 Experience Conclusions

Throughout this experience, we generated, with the help of ChatGPT, a set of 63 rules that can be consulted in Appendix I (p. 129). Within this set are rules for various types of conflicts: loops, redundancies, and rules with the same trigger but different actions on the same actuator (inconsistencies).

Source	Target	Type	Id
state_0	state_20	Directed	edge_rule_46_state_0_state...
state_1	state_21	Directed	edge_rule_46_state_1_state...
state_2	state_20	Directed	edge_rule_46_state_2_state...
state_3	state_21	Directed	edge_rule_46_state_3_state...
state_4	state_20	Directed	edge_rule_46_state_4_state...
state_5	state_21	Directed	edge_rule_46_state_5_state...
state_6	state_20	Directed	edge_rule_46_state_6_state...
state_7	state_21	Directed	edge_rule_46_state_7_state...
state_8	state_20	Directed	edge_rule_46_state_8_state...
state_9	state_21	Directed	edge_rule_46_state_9_state...
state_0	state_20	Directed	edge_rule_61_state_0_state...
state_1	state_21	Directed	edge_rule_61_state_1_state...
state_2	state_20	Directed	edge_rule_61_state_2_state...
state_3	state_21	Directed	edge_rule_61_state_3_state...
state_4	state_20	Directed	edge_rule_61_state_4_state...
state_5	state_21	Directed	edge_rule_61_state_5_state...
state_6	state_20	Directed	edge_rule_61_state_6_state...
state_7	state_21	Directed	edge_rule_61_state_7_state...
state_8	state_20	Directed	edge_rule_61_state_8_state...
state_9	state_21	Directed	edge_rule_61_state_9_state...

Figure 6.16: Edges associated with rules 46 and 61

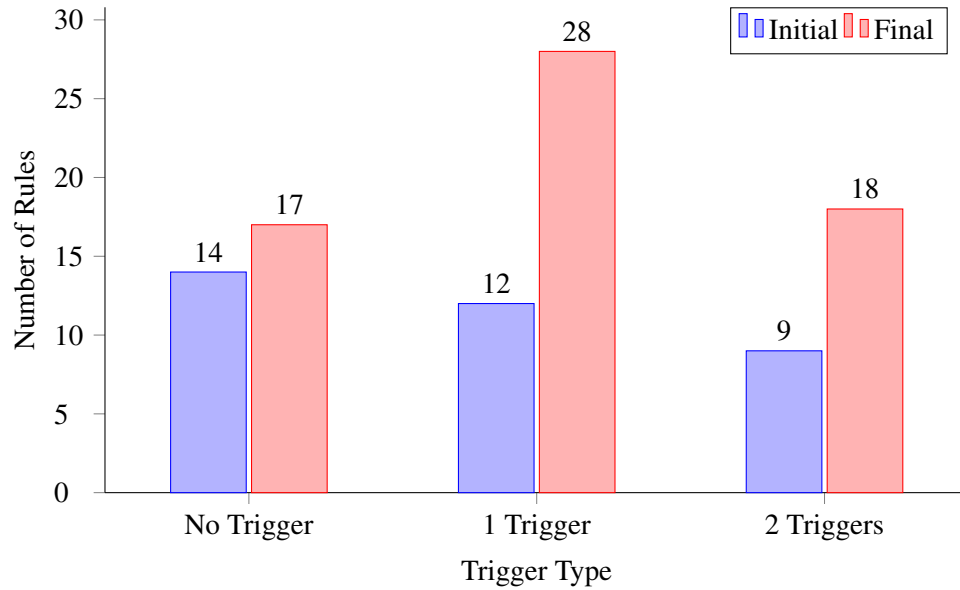


Figure 6.17: Comparison of the number of rules per trigger type between initial and final sets

Analyzing this final set of rules, it is possible to conclude that most of the rules generated from the initial rules contained one or two triggers (*cf.* Figure 6.17). Although there is some difference between the number of rules with one trigger and the other types in the final rule set, the number of rules of each type of trigger is sufficient to allow several comparisons between rules with different types of triggers.

On the other hand, when analyzing the distribution of the rules by the type of periodicity they

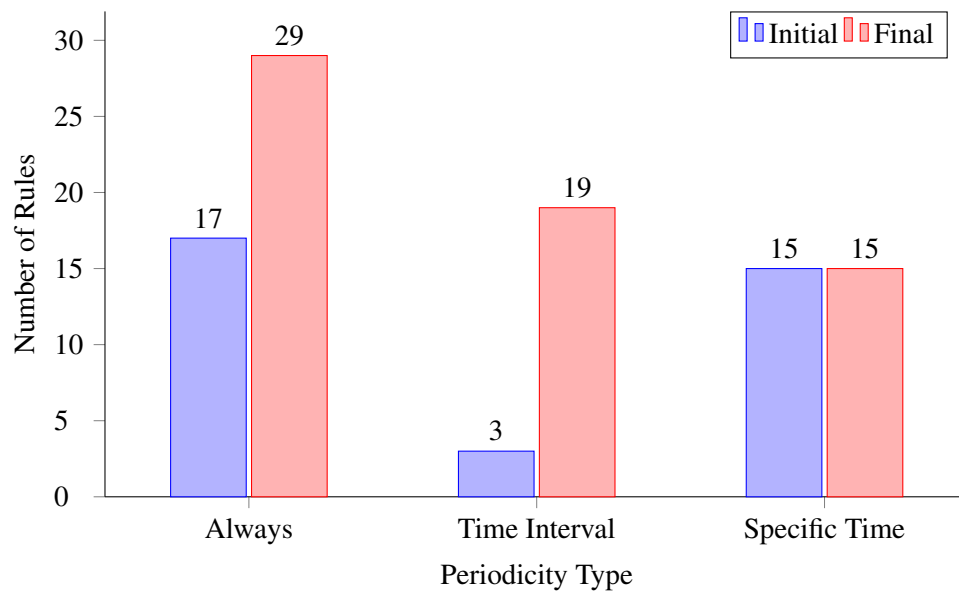


Figure 6.18: Comparison of the number of rules per periodicity type between initial and final sets

present, in Figure 6.18, we conclude that what was initially identified as a possible problem — the fact that there are few rules with periodicity “Time Interval” — turned out not to be influential. In the new rules generated from the initial set, we verified that the number of rules with this type of periodicity grew enough to compensate for the initial discrepancy. Thus, although the periodicities are not equally distributed among the rules, we verify that the number of rules with each type of periodicity is sufficient to test different combinations.

As mentioned at the end of the redundancy analysis, the detected redundancies only fit 2 of the 16 possible combinations for the Rule Triggers / Rules Periodicity pair (*cf.* Table 6.2, p. 69). Although we wish to explore more types of combinations in this experiment, all redundancies detected by our system were effectively redundancies. In addition, we could detect more conflicts of this type than ChatGPT and with a higher hit rate since ChatGPT also detected a pair of redundant rules that proved invalid. These are good indicators that our system is correctly detecting these conflicts.

Moving on to the analysis of inconsistencies, we resolved cases of divergence between the decision of our system and the detection made by ChatGPT by checking the rules and the graphs where they were inserted. As with redundancies, our system proved to be highly effective in detecting this type of conflict, detecting more inconsistencies than ChatGPT. All the inconsistencies that we detected were correct. However, as we verified in Table 6.4 (p. 78), not all Rule Triggers / Rules Periodicity pair combinations were explored. Even so, we must emphasize that we identified inconsistencies for all types of rule triggers and for 5 of the 8 types of periodicity rules.

Finally, regarding loops, we demonstrated that our system can detect loops containing three, four, or five rules. We also prove that our system can detect the combinations of rules that form loops in the same path, despite only returning one of these combinations.

6.2.2 Extensive Analysis of the Detected Conflicts

As we concluded at the end of the previous experience, although the results were quite positive, we still needed to explore many cases concerning redundancies and inconsistencies.

So, to know concretely all the situations in which our system can detect redundancies and inconsistencies, we conducted a more extensive analysis. To do so, we will explore all possible combinations of triggers, actions, and periodicities and check which cases are correctly detected by our system, which ones should be detected and are not, and which ones should not be detected but are detected.

To generate all possible combinations of rules, we created a script so that the different combinations of rules could be easily created in an iterative and fast way, varying the following parameters:

- **Rules' triggers and sensors used:**

- rules sharing the same sensors: all sensors used in the triggers of a rule are also used in the triggers of the other rule;
- rules sharing only part of the sensors: some of the sensors used in the triggers of one rule are also used in the triggers of the other rule;
- one of the rules has no trigger part;
- both rules have no trigger part.

- **Sensors' type comparison:**

- *boolean* vs. *boolean*;
- *numeric* vs. *numeric*.

- **Comparison symbol between the sensor and attributed value in trigger:** $=, \neq, <, >, \leq, \geq$;

- **Comparison between values attributed to each sensor:**

- $value1 = value2$;
- $value1 < value2$;
- $value2 < value1$;
- $value1 \neq value2$.

- **Actuator's type:**

- *boolean*;
- *numeric*.

- **Rules' action comparison:**

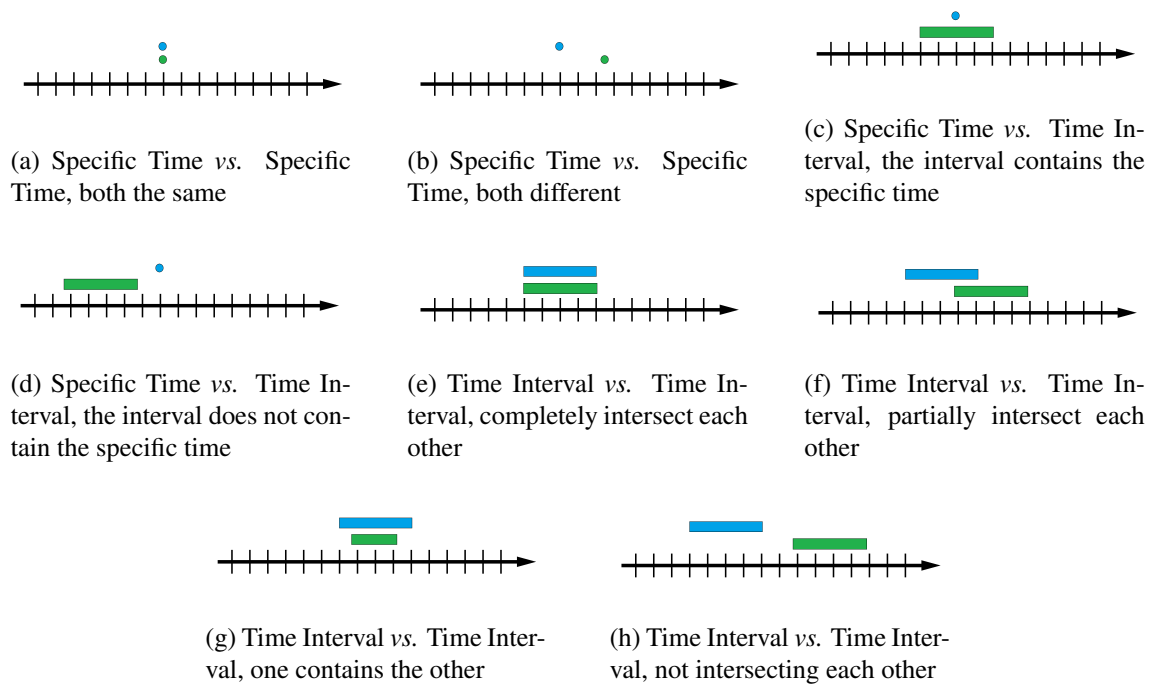


Figure 6.19: Rules' periodicity comparison

- rules have the same action in the same actuator;
- rules have different actions in the same actuator.

- **Comparison between the periodicity of each rule:**

- Always vs. Always;
- Always vs. Specific Time (*e.g.*, “Daily at 16:30”);
- Always vs. Time Interval (*e.g.*, “Every Tuesday between 16:00 and 18:00”);
- Specific Time vs. Specific Time, both the same (*cf.* Figure 6.19a);
- Specific Time vs. Specific Time, both different (*cf.* Figure 6.19b);
- Specific Time vs. Time Interval, the interval contains the specific time (*cf.* Figure 6.19c);
- Specific Time vs. Time Interval, the interval does not contain the specific time (*cf.* Figure 6.19d);
- Time Interval vs. Time Interval, completely intersect each other (*cf.* Figure 6.19e);
- Time Interval vs. Time Interval, partially intersect each other (*cf.* Figure 6.19f);
- Time Interval vs. Time Interval, one contains the other (*cf.* Figure 6.19g);
- Time Interval vs. Time Interval, not intersecting each other (*cf.* Figure 6.19h).

To create the rules, we used three sensors and two actuators. Two of the sensors created are of the *numeric* type, and the other is of the *boolean* type. Regarding the actuators, there was one of

each type. There was no relationship between the sensors and actuators. That is, a change in the state of the actuator did not directly imply any change in the state of any of the existing sensors.

For the case of *rules sharing the same sensors*, each of the rules of the initial pair created contained only one trigger, in which we used the same sensor in both triggers. In the case of *rules sharing only part of the sensors*, one of the rules of the initial pair created contained two triggers, one of the used sensors being the same as the one used by the other rule. Regarding the trigger that did not use the same sensor as the trigger of the other rule, it was always the same in all rules with two generated triggers, *i.e.*, it used the same sensor (the one of type *numeric* extra), the same comparison symbol and the same value. We based this decision on the fact that this trigger does not influence the comparison between the triggers of the rules — it only demonstrates that we can distinguish the cases of rules that share all the sensors from the cases in which they share only some. For situations where *one of the rules has no trigger part*, one of the rules of the pair created did not contain a trigger, and the other used one of the previously presented sensors, according to the type of sensors being analyzed. Neither sensor was used for the remaining case where *both rules have no trigger part*.

Among the common devices used in triggers, there are two possible situations: the two sensors are of the *numeric* type or the *boolean* type. It makes no sense to compare *boolean* vs. *numeric* since the same sensor can only be one of the types.

Depending on the type of sensor used, the symbols used to assign the sensor values to which the trigger will be executed vary. If the sensor is of the *boolean* type, use $=$ and $\neq$. If the sensor is of the *numeric* type, it is possible to use all the symbols. Thus, from the pair of rules initially generated, we get a total of:

- 4 rules, if both rules have triggers and the common sensors are of the *boolean* type;
- 36 rules, if both rules have triggers and the sensors used are of the *numeric* type;
- 3 rules, if one of the rules does not have triggers and the sensor used in the other is of type *boolean*;
- 7 rules, if one of the rules has no triggers and the sensor used in the other is of the *numeric* type;
- 2 rules, if both rules have no triggers.

To assign a value to trigger conditions, it is first necessary to form all possible pairs between the rules (*e.g.*, join two rules with the symbol $<$, join a rule with the symbol $>$ with another with the symbol $\geq$, join a rule without trigger with one with the symbol $=$, among others). It is necessary to remember that the common sensors must have the same type. The possibilities for the value of the conditions of each pair vary according to the symbols used. In case the symbols used in the two triggers are $=$ or $\neq$, then the comparison values used will be just $value1 = value2$ and $value1 \neq value2$. Otherwise, all comparison values will be used except for the case $value1 \neq value2$. The values assigned to each sensor in each trigger are random among the values at which the sensor

works, respecting these conditions. From the rules generated with the previous condition, we get a total of:

- 8 pairs of rules, if both rules have triggers and the common sensors are of type *boolean*;
- 104 pairs of rules, if both rules have triggers and sensors used are of type *numeric*;
- 2 pairs of rules, if one of the rules has no triggers and the sensor used in the other is of *boolean* type;
- 6 pairs of rules, if one of the rules has no triggers and the sensor used in the other is of the *numeric* type;
- 1 pair of rules, if both rules have no triggers.

The penultimate step is to assign actions to each pair of rules. The rules of each pair can have the same action on the same actuator or else different actions on the same actuator. The actuator, just like the sensors, can have two different types (*boolean* or *numeric*). In each rule pair, the actuator is the same. Thus, the number of rule pairs quadruples at the end of this step.

Regarding the periodicities, we obtained ten rules for each pair formed. Considering the pair (rule1, rule2), we distribute the periodicities as follows to obtain all possible combinations:

- rule1 with periodicity Always;
- rule2 with periodicity Always;
- rule2 with a specific time as periodicity (*e.g.*, Daily at 16:30) ;
- rule2 with a time interval as periodicity (*e.g.*, Daily between 16 and 19);
- rule1 with a specific time that intersects the interval of rule2 and is equal to the specific time defined for rule2 as periodicity (*e.g.*, Daily at 16:30);
- rule2 with a specific time, different from the one in rule1, as periodicity (*e.g.*, Daily at 10);
- rule2 with a time interval that does not contain the specific time defined for rule1 as periodicity (*e.g.*, Daily between 12 and 14);
- rule1 with a time interval that totally intersects one of those that was defined for rule2 as periodicity (*e.g.*, Daily between 16 and 19);
- rule2 with a time interval that partially intersects the one defined for rule1 as periodicity (*e.g.*, Daily between 18 and 21);
- rule2 with a time interval contained in the one defined for rule1 as periodicity (*e.g.*, Daily between 16 and 17).

Next, we will analyze, for each of the types of conflicts under analysis — inconsistencies and redundancies —, which combinations correspond to conflicting situations and how many of these our system can detect.

6.2.2.1 Extensive Analysis for Inconsistencies

As already mentioned, in our work, we only focus on one of the types of inconsistencies: those caused by rules that contain the same triggers but different actions on the same actuator. Based on this, it is necessary to identify which combinations of parameters defined in Section 6.2.2 (p. 87) fit this type of inconsistency, *i.e.*, in which cases rules are triggered simultaneously and influence the same actuator differently.

Starting by analyzing the triggers of the rules and the sensors used, all the presented cases apply to this type of conflict. It applies when the rules share all the sensors because they will trigger in the same situations if they have similar conditions. The case in which the rules share only part of the sensors also applies because, at a time when the conditions defined for all the remaining sensors used in the rules are valid, if the conditions defined for the sensors shared in both rules are similar, then the rules will run at the same time. In case one or two rules do not have a trigger, it is also valid because rules that do not have triggers will be executed at the same time as other rules in case the periodicities intersect at some point.

Regarding the valid combinations between the type of sensors, the comparison symbols, and the values assigned to each sensor for rules sharing the same sensors (or part), they were previously defined in Table 5.1 (p. 52).

Moving on to the actions part, we are interested in the rules having different actions in the same actuator since the two existing actuator types apply to inconsistencies.

Lastly, the periodicity comparisons that we are interested in exploring are those that allow rules to be executed simultaneously at some point:

- Always *vs.* Always;
- Always *vs.* Specific Time;
- Always *vs.* Time Interval;
- Specific Time *vs.* Specific Time, both the same;
- Specific Time *vs.* Time Interval, the interval contains the specific time;
- Time Interval *vs.* Time Interval, completely intersect each other;
- Time Interval *vs.* Time Interval, partially intersect each other;
- Time Interval *vs.* Time Interval, one contains the other.

In this way, all pairs of rules that fit these combinations are considered inconsistent. It remains to be seen how many of these cases our system can detect.

After running the rule sets generated in our system and checking all the conflicts detected by it, comparing them with the pairs previously identified as being inconsistent, we concluded that we could detect all possible conflicting combinations.

As we analyzed the results obtained by running the sets of rules generated in our system, we could identify a pattern regarding how the identification of inconsistencies in the generated graphs varied according to the type of actuator. While the inconsistencies whose actuator type in the rules is *numeric* are identified in the graph as edges that start from the same node but reach different final nodes (there are cases in which, in addition to this identification, they can also be identified as loops in the graph), rules whose actuator is of type *boolean* are uniquely identified through loops in the graph. We can easily explain this situation.

Focusing on the first part of this pattern, *i.e.*, when the actuator is of type *numeric*, let us look at the following pair of rules:

Rule 1: If the living room door is open, set the living room lights brightness to 50%.

Rule 2: If the living room door is open, turn off the living room lights.

In an initial scenario where the living room lights are off, and the living room door is open, then Rule 1 will be executed (Rule 2 is not executed for now as the lights are already off), turning the lights on of the living room at 50% brightness. Then Rule 2 will be executed, turning off the lights. Under these conditions, Rule 1 will be executed again and then Rule 2 again, repeating this cycle continuously. So here we have a cycle in the graph. As the rules have the same trigger and act on the same actuator with different actions simultaneously, we consider this conflict an inconsistency.

However, if, in the initial scenario, the living room lights are turned on with brightness other than 50% and the living room door is open, a loop will no longer occur. In this case, the rules will be identified as inconsistent because the corresponding edges start from the same initial node of the graph and have different nodes as the destination. A smart home cannot simultaneously be in two different scenarios, so conflict exists.

Having explained the pattern's first part, let us focus on the second part, relative to when the actuator is of type *boolean*. To do so, consider the following rules:

Rule 1: If the living room door is closed, turn on the living room TV.

Rule 2: If the living room door is closed, turn off the living room TV.

This situation is very similar to the first scenario explored above. Considering an initial environment where the living room door is closed and the television is on, Rule 2 will execute. Rule 1 will not initially execute because the television is already on. When Rule 2 executes, the TV turns off in the environment under analysis. In this situation, Rule 1 will then execute, turning the TV back on, which serves as a trigger for a new execution of Rule 2. Therefore, there will be a continuous change between these two scenarios, meaning the graph has a loop. Again, as the rules have the same trigger and act on the same actuator with different actions simultaneously, we consider this conflict an inconsistency.

The results allow us to state that our system can detect inconsistencies between all existing combinations among the defined parameters in the previous environment described in Section 6.2.2 (p. 87). On the other hand, the system did not mistakenly detect any conflicts, which allows us to conclude that it has demonstrated maximum effectiveness.

6.2.2.2 Extensive Analysis for Redundancies

The second type of conflict we will explore in this extensive analysis is redundancies. Retrieving the redundancy definition previously provided in Section 5.2.5.3 (p. 55), two rules are redundant when both define the same trigger and simultaneously trigger the same action. Redundant rules do not add any value to the system. Based on this definition, it is necessary to identify which parameters defined in Section 6.2.2 (p. 87) we can use to identify all combinations of rules that represent redundancies.

Only *rules sharing the same sensors* and *rules that have no trigger part* can be considered redundant. These cases allow the triggers to overlap by setting the same conditions. The remaining hypotheses do not apply to this type of conflict, as the rules are only executed simultaneously in some situations.

We previously described the acceptable combinations of sensor type, comparison symbols, and values assigned to each sensor in Table 5.2 (p. 56).

Moving on to the actions, we are concerned with the rules that have the same action in the same actuator, as the two existing actuator categories apply to inconsistencies.

Finally, the periodicity comparisons we are interested in are those that allow rules to be run concurrently at some point:

- Always vs. Specific Time;
- Always vs. Time Interval;
- Specific Time vs. Time Interval, the interval contains the specific time;
- Time Interval vs. Time Interval, partially intersect each other;
- Time Interval vs. Time Interval, one contains the other.

As a result, any rule pairings that meet these combinations are deemed inconsistent. Periodicities *Always vs. Always*, *Specific Time vs. Specific Time, both the same*, and *Time Interval vs. Time Interval, completely intersect each other* would also be valid possibilities. However, rules with the same triggers, action, and periodicity are equal rules. Therefore, the system initially eliminates one, so these three periodicities do not apply. How many of these instances our system can detect remains to be seen.

We determined that we could discover all conceivable conflicting combinations after executing the rule sets created in our system and comparing them with the pairings previously recognized as contradictory.

For redundancies, contrary to what happens with inconsistencies, there is only one way to detect them in graphs: edges that leave the same initial vertex towards the same final vertex, *i.e.*, parallel edges. As we can conclude from the definition, redundant rules are rules with the same triggers and actions executed simultaneously. Based on that, it makes sense to represent the redundancies through parallel edges in the graph.

The acquired findings allow us to assert that in the previously stated environment in Section 6.2.2 (p. 87), our system can identify redundancies between all existing combinations of the given parameters. On the other hand, no conflict was recognized by mistake, allowing us to infer that the system has performed with a maximum success rate.

6.3 Effectiveness of the Implementation of Connected Components Algorithm

The last analysis we will carry out intends to analyze the metrics obtained using our system with the connected components algorithm implemented compared with the metrics we obtain when this algorithm is not implemented in our system.

For such an analysis, we chose, from the ruleset generated during the Section 6.2.1 (p. 61) (*cf.* Appendix I, p. 129), sets of 5, 10, 15, 20, 30, 40, and 50 rules in a random way, with the rules of the set of 5 rules being in the set of 10 rules (the same applies to the remaining sets). We then ran our system (with and without the connected components algorithm), measuring the total execution time, the time it took to detect redundancies, inconsistencies, loops, and the total number of nodes, edges, devices, and graphs. The obtained results can be found in Table 6.6 and Table 6.7 (p. 95).

Number of Rules	5	10	15	20	30	40	50
Total Time (ms)	124	200	285	450	647	1206	1474
Total Time Redundancies (ms)	24	28	30	32	36	45	53
Total Time Inconsistencies (ms)	3	4	7	11	22	35	60
Total Time Loops (ms)	4	6	8	10	12	54	100
Total Number of Nodes	17	47	74	118	160	256	317
Total Number of Devices	6	19	28	32	40	48	50
Total Number of Edges	16	25	53	69	126	226	353
Total Number of Graphs	3	8	11	13	14	16	16

Table 6.6: Metrics obtained using our system with connected components algorithm

Table 6.7 (p. 95) only contains results for sets of 5 and 10 rules, and incompletely, as the system spent 24 hours running for the set of 10 rules and, in that time, all the edges corresponding to the rules had not yet been generated. For the remaining sets of rules, obtaining the number of nodes was impossible because we got a fatal error: “JavaScript heap out of memory”. This error is related to the fact that there are many possible scenarios in which all devices used in these sets of rules can be found.

Number of Rules	5	10
Total Time (ms)	531	n/a
Total Time Redundancies (ms)	186	n/a
Total Time Inconsistencies (ms)	32	n/a
Total Time Loops (ms)	18	n/a
Total Number of Nodes	72	62207
Total Number of Devices	6	19
Total Number of Edges	132	n/a
Total Number of Graphs	1	1

Table 6.7: Metrics obtained using our system without connected components algorithm

Focusing only on the two sets we can compare, we verify that the system with the connected components algorithm implemented is much more efficient than the system without this algorithm. Comparing the results obtained for the set with 5 rules, the number of nodes generated by the system that does not contain the implemented algorithm is about four times higher than the number of nodes generated in the other system. As the number of nodes is greater, there are also about eight times more edges because, for each rule, there are more scenarios with which it can be associated. With this discrepancy between the number of nodes and edges, as expected, the time it takes the system to find the conflicting rules of each type is also much higher. To detect redundancies and inconsistencies, the system with the connected components algorithm needs to perform a substantially lower number of comparisons between edges that come out of the same node than those performed in the other system. The difference is exorbitant if we move forward to the set of 10 rules, comparing only the number of generated nodes. For the same number of devices (19), 47 nodes distributed over 8 different graphs are generated in the system with the connected components algorithm implemented, while in the other 62207 nodes are generated and contained only in one graph.

In this way, we demonstrate the importance of implementing the connected components algorithm in our system. Without it, detecting conflicts in more complex smart environments would be very difficult. Thus, the implementation of this algorithm is essential so that our system can have reduced execution times and can be used in environments with many rules and devices without compromising the total execution time or facing any limitation derived from the number of scenarios generated.

Still focusing on the variation in the number of generated nodes according to the number of devices, we verified that its curve resembles the curve presented by an exponential function fitted to the existing data. As we can observe in Figure 6.20 (p. 96), the data points align closely with the exponential curve. This close alignment suggests that the total number of nodes exhibits exponential growth concerning the number of devices used.

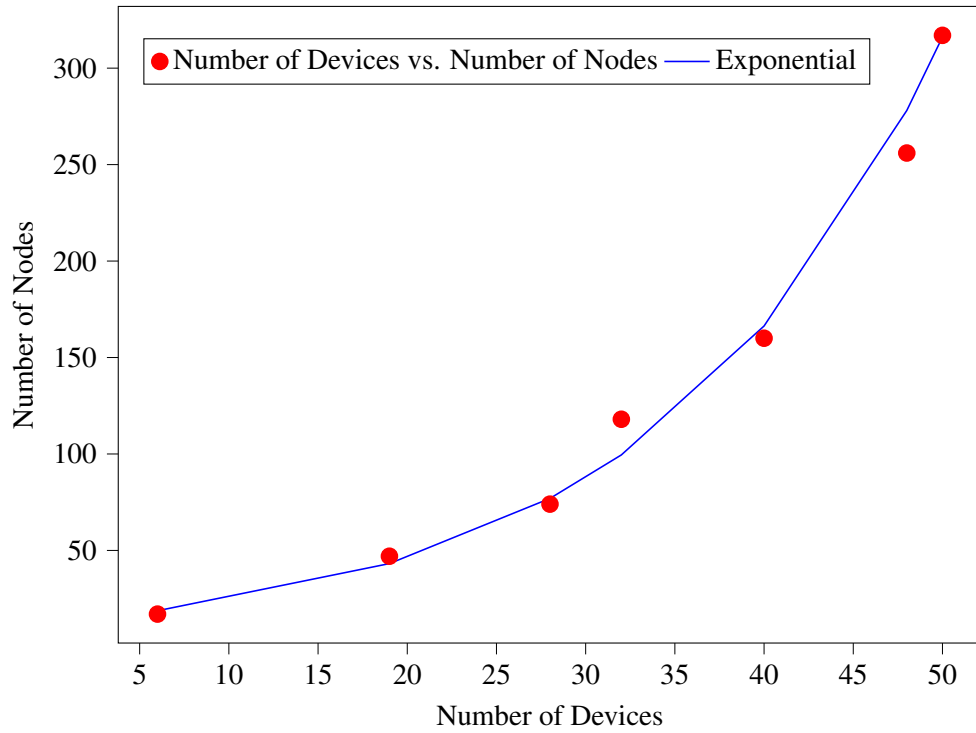


Figure 6.20: Comparison of Number of Devices vs. Number of Nodes with Exponential Function

6.4 Conclusions

Through Experiment 1 (*cf.* Section 6.2.1, p. 61), we demonstrated that our system could detect conflicts in environments similar to those found in smart homes in everyday life. Although in this experiment we could not explore all possible combinations for conflicts between rules and the sensors used, it was possible to detect loops between 3, 4, and 5 rules, as well as inconsistencies between rules with the same trigger but different actions on the same device, and redundancies.

In Experiment 2 (*cf.* Section 6.2.2, p. 87), through extensive analysis, we could concretely discover which combinations of rules and sensors in the triggers our system can detect as conflicting. The results were positive since we could detect inconsistencies and redundancies in all supposed cases. Thus, we demonstrate that our system is working as expected.

Finally, we could positively demonstrate the advantage of implementing the connected components algorithm in our system (*cf.* Section 6.3, p. 94), as it was much more efficient and allowed conflict detection in environments with more rules and devices than the system without this improvement.

6.5 Validity Threats

For this evaluation and validation phase, it is important to highlight potential limitations that may impact the effectiveness of the validation process. The limitations we were able to identify are the

following:

Volatility in the answers given by ChatGPT. The answers given by ChatGPT tend to vary, which can mean that the conflicts detected or not by this chatbot may be different if we ask again to detect conflicts between rules. Although we repeated the conflict detection request at least three times in order to try to reduce as much as possible the possible discrepancies between responses that may exist, it is not guaranteed that there were no conflicts left to be detected.

Devices chosen in Experiment 2. As mentioned in Section 6.2.2 (p. 87), for all generated rules, we used the same environment that contained three sensors (2 of type *numeric* and one of type *boolean*) and two actuators (one of each type), with no relationship between them. In this setup, it was possible to detect conflicts in all expected situations. However, in case there is any relationship between the devices or the extra trigger in the case of *rules sharing only part of the sensors* has a different configuration (in our experience, we used the same extra trigger in all rules with two triggers), it is not insured that the results obtained will be the same. Carrying out other experiments varying these identified factors will be a possible way to explore the various possible situations and validate the conclusions for all environments, regardless of the chosen devices.

Throughout the validation and evaluation process, we minimize the impact of these threats. However, it is necessary to keep in mind that they still exist.

6.6 Summary

This chapter explains the experiments carried out to evaluate and validate our system and hypothesis. We evaluated the results obtained in the experiments and drew some conclusions.

In Experiment 1 (*cf.* Section 6.2.1, p. 61), we demonstrated that our system could be used in real environments. For this, we design a smart environment like a smart home containing sensors and actuators and create rules similar to those used in these environments daily. The results made us conclude that our system can detect loops, inconsistencies, and redundancies under these conditions.

Through Experiment 2 (*cf.* Section 6.2.2, p. 87), we could know concretely all the combinations of rules in which our system can detect inconsistencies and redundancies. The results were quite positive, detecting conflicts in all foreseen situations.

Finally, we compared the system's performance with and without implementing the connected components algorithm (*cf.* Section 6.3, p. 94), obtaining results quite favorable to the implementation of this algorithm in our solution.

After carrying out all the experiments, we draw some conclusions in Section 6.4 (p. 96), and then we present the potential limitations that may impact the effectiveness of our validation process (*cf.* Section 6.5, p. 96). Although we have tried to minimize the potential impact of these threats, we must be aware that they exist and can influence the results obtained.

Chapter 7

Conclusions

7.1	Conclusions	98
7.2	Challenges	101
7.3	Future Work	101

In this chapter, we will reflect on the work carried out throughout this dissertation. To this end, in Section 7.1, we will revisit the *research questions* created and our hypothesis. Next, in Section 7.2 (p. 101), we identify the main challenges we face during this work. Finally, in Section 7.3 (p. 101), we suggest possible future improvements to our system.

7.1 Conclusions

Over the last few years, there has been significant growth in the sale and use of IoT devices. One of the most relevant applications of these devices is the creation of automation in different smart spaces, such as homes. With the increasing number of smart devices in each person's home, the number of trigger-action rules designed to get the most out of the devices has also risen.

With the increase in the number of existing rules in a smart environment, and considering that many people use these devices without great programming knowledge, there is a greater chance of conflicts between the created rules without users correctly understanding what is happening. Under these conditions, it becomes essential to have systems that help end-users to control the smart environment and manage existing rules, identifying potential conflicts before they occur.

During the study of the state of the art that we carried out (*cf.* Chapter 3, p. 8), we verified that there is a lack of solutions that explore the detection of conflicts in smart environments, exploring all the possible scenarios of combinations between all the existing devices. Furthermore, most of the identified solutions were not prepared to be used in multi-user environments and, of those that were, did not take privacy issues into account. If we focus on the conflict resolution part, it relies on one-time exceptions most of the time.

Based on the previously identified problems, we built a *desiderata* (*cf.* Section 4.2, p. 33) where we set out the goals we intended to achieve with this work. Throughout this dissertation,

we suggest a possible system capable of solving some of the identified conflict detection problems. Thus, based on the flaws detected in the conflict detection systems in smart environments analyzed, we defined the hypothesis that guided our work:

“Can a graph-based approach, utilizing nodes to represent all possible scenarios of an Internet of Things (IoT) system, and edges to illustrate the transitions between scenarios driven by the rules, effectively and efficiently detect inconsistencies, loops, and redundancies within the system?”

To help us guide our research so that we can confirm or refute our hypothesis, we defined seven research questions (*cf.* Section 4.4, p. 35) that we now retrieve to answer them:

RQ1. *Is it possible to build a system for a smart home context capable of **generating every possible scenario** containing the states in which intelligent devices can be found in that environment?*

Yes, it is possible. As we saw in Section 5.2.2 (p. 42), after all existing devices in a given smart space are defined in a YAML file, as well as the respective rules, we can obtain all possible valid combinations for the states of all sensors and actuators.

RQ2. *Is it possible to build a system for a smart home context capable of **reducing the number of possible scenarios** to make it more efficient?*

As stated in Section 5.2.2 (p. 42), groups of devices that relate to each other and follow the same rules tend to emerge in smart environments. This way, if we build an undirected graph where each node represents a different device and the edges connect devices used in the same rules, then, applying the connected components algorithm, we can get the groups of strongly connected devices of that environment. Based on this information, instead of generating just one graph containing the possible scenarios for all devices, we generate a different graph for each of the groups obtained. Thus, we managed to reduce the number of possible combinations between the device states since, in each graph, there are only scenarios containing devices that influence each other.

We tested the efficiency of this solution in Section 6.3 (p. 94), with very positive results.

RQ3. *Is it possible to build a system for a smart home context capable of **predicting loop conflicts** between three or more rules?*

Taking advantage of the graph-based approach, after creating the edges between the graph vertices, to discover loops between three or more rules, it is only necessary to resort to a depth-first search (*cf.* Section 5.2.5.1, p. 47).

We validated this solution through Section 6.2 (p. 61).

RQ4. *Is it possible to build a system for a smart home context capable of **predicting conflicts** between rules with (partially) the same triggers but different actions in the same device?*

In a graph, we can detect this conflict when a pair of outgoing edges departs from the same initial node and has different nodes as their destination, which is impossible in a smart environment. This conflict is also detected, as mentioned in Section 5.2.5.2 (p. 50), when there is a cycle between two rules in one of the directed graphs and these rules share part of the sensors or when at least one of the rules has no trigger.

We validated this solution through Section 6.2 (p. 61).

RQ5. *Is it possible to build a system for a smart home context capable of predicting **conflicts caused by redundant rules**?*

Using the particularities of graphs again, we can observe that in a multigraph, when two parallel edges are present, it indicates rules executed under the same conditions and with the same action. Thus, if the triggers of these rules overlap, then these rules are redundant (cf. Section 5.2.5.3, p. 55).

We validated this solution through Section 6.2 (p. 61).

RQ6. *Is it possible to build a system for a smart home context capable of predicting different conflict types and **takes into account each rule periodicity** when deciding if a set of rules contains conflicts?*

We can specify the time and frequency with which each rule should be executed using a cron expression. Thus, when the system detects a conflict between rules in a graph, it checks whether their periodicities intersect (cf. Section 5.2.4, p. 46). Only if this intersection occurs are the rules effectively in conflict.

RQ7. *Is it possible to build a system for a smart home context capable of **providing explanations about the existing conflicts** to the users so that they can easily understand why a certain conflict occurs?*

Our system can present the users with a small explanation about why there is a conflict in case of redundancies and inconsistencies.

Through the answers to the research questions, we observed that we could answer affirmatively to all of them. Additionally, and taking into account these affirmative responses obtained, we were able to confirm our hypothesis. It is effectively possible to detect loops, inconsistencies, and redundancies using a graph-based approach efficiently.

Overall, our system achieved the goals defined in **D1**, **D2**, and **D3** in our *desiderata*. We partially achieved **D4** as we do not provide explanations about loop conflicts. Although we initially aimed to solve some more of the problems we identified, as shown by points **D5** and **D6** of *desiderata*, it was not possible to explore and find solutions for them due to time constraints and the complexity that these problems involve. Developing a solution from scratch to detect and resolve conflicts in environments with multiple users is not simple. With that in mind, we preferred to focus our work on just one of the detected issues to find a quality solution rather than building a system that covered all detected problems superficially.

In conclusion, our system is an evolution compared to existing conflict detection systems. Although there are improvements that must be made to correct the currently existing limitations (*cf.* Section 5.3, p. 58), this system demonstrated the feasibility of using a graph-based approach to represent all possible scenarios between device states in an IoT system and detect conflicts in smart environments with different devices and multiple rules, as long as implemented together with the connected components algorithm.

7.2 Challenges

During this project's realization, we faced some challenges mainly related to validating our system. These problems were mostly due to the interaction with ChatGPT.

Sometimes, it is complex to know the best way to explain what we want in a straightforward way so that ChatGPT returns what we desire in response. Throughout the validation phase, especially in the initial rule generation phase, it took some time to understand the correct way to place orders and what details we needed to provide to ChatGPT.

Furthermore, making this chatbot realize that the answers provided contain errors or do not adapt to the described situation can be challenging.

Finally, the volatility of the answers provided by ChatGPT for the same request forced us to repeat specific requests several times until the answers started to resemble more.

7.3 Future Work

The system we developed also contains a set of limitations that must be taken into account (*cf.* Section 5.3, p. 58). As the construction of conflict detection systems using the graph-based approach still has much to explore, several ways exist to upgrade this system. Some of them are as follows:

Fix previously identified limitations. As identified in Section 5.3 (p. 58), our implementation has some constraints which make our system prevent our system from accepting certain types of rules or devices. It is crucial to resolve these limitations to allow the system to be used in environments with more complex rules or to detect more types of conflicts.

Detect conflicts between rules defined by different users, taking into account privacy concerns. As we saw, more and more people are using IoT devices nowadays. Typically, these devices are used in environments with multiple users, such as homes or offices, interacting with the devices. Therefore, this type of system must be able to detect conflicts in multi-user environments and consider the privacy of other users and the rules created by each one. Thus, each user should only be able to access their created rules. If there are conflicts between their rules and others created by another user, they should have access to as little information as possible about the conflicting rules.

Find other algorithms to reduce the number of possible scenarios. Although we have significantly reduced the number of existing nodes with the connected components algorithm, scenarios are still generated with no associated rule. As long as no rules in the environment are associated with them, these states must be ignored. On the other hand, some nodes represent equivalent scenarios and, therefore, can also be ignored. This way, we can further reduce the number of nodes and make the system more efficient.

Build a graphical interface. A graphical interface where users can interactively define rules and receive notifications and explanations of existing conflicts will make the system more attractive. Nowadays, a good GUI attracts many users.

Expand the system to suggest ways to solve the detected conflicts. The conflict resolution task sometimes becomes complicated for inexperienced users with IoT equipment and environments, especially when involving several rules. Thus, adding the functionality to suggest possible resolutions for existing conflicts will facilitate this task and increase the ease of creating and living in smart environments.

By integrating these suggested improvements, the system's efficiency, performance, usability, and functionalities will improve significantly, allowing it to become a complete system capable of being used by anyone in the most diverse environments.

References

- [1] Muhammad Raisul Alam, Mamun Bin Ibne Reaz, and Mohd Alauddin Mohd Ali. A Review of Smart Homes—Past, Present, and Future. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1190–1203, 2012.
- [2] Ahmed S. Alfakeeh, Ali H. Al-Bayatti, Francois Siewe, and Thar Baker. Agent-based negotiation approach for feature interactions in smart home systems using calculus of the context-aware ambient. *Transactions on Emerging Telecommunications Technologies*, 33(2), 2019.
- [3] Cambridge University Press & Assessment. Cambridge Dictionary IoT Definition. <https://dictionary.cambridge.org/dictionary/english/internet-of-things>. [Online; Accessed 2023].
- [4] AWS. Amazon AWS IoT Definition. <https://aws.amazon.com/what-is/iot/>. [Online; Accessed 2023].
- [5] Lei Bu, Wen Xiong, Chieh-Jan Mike Liang, Shi Han, Dongmei Zhang, Shan Lin, and Xuan-dong Li. Systematically Ensuring the Confidence of Real-Time Home Automation IoT Systems. *ACM Transactions on Cyber-Physical Systems*, 2(3):1–23, 2018.
- [6] Inc Cisco Systems. Cisco Annual Internet Report (2018–2023) White Paper. <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>, 2020. [Online; Accessed 2023].
- [7] Sven Coppers, Davy Vanacken, and Kris Luyten. FORTNIoT: Intelligible Predictions to Improve User Understanding of Smart Home Behavior. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(4):1–24, 2020.
- [8] Sven Coppers, Davy Vanacken, and Kris Luyten. FortClash: Predicting and Mediating Unintended Behavior in Home Automation. *Proceedings of the ACM on Human-Computer Interaction*, 6(EICS):1–20, 2022.
- [9] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. Empowering End Users in Debugging Trigger-Action Rules. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2019.
- [10] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. *My IoT Puzzle: Debugging IF-THEN Rules Through the Jigsaw Metaphor*, pages 18–33. Springer International Publishing, 2019.
- [11] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. A Semantic Web Approach to Simplifying Trigger-Action Programming in the IoT. *Computer*, 50(11):18–24, 2017.

- [12] João Pedro Dias, Bruno Lima, João Pascoal Faria, André Restivo, and Hugo Sereno Ferreira. Visual Self-Healing Modelling for Reliable Internet-of-Things Systems. In *Computational Science – ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part V*, page 357–370, Berlin, Heidelberg, 2020. Springer-Verlag.
- [13] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Empowering Visual Internet-of-Things Mashups with Self-Healing Capabilities. In *2021 IEEE/ACM 3rd International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, pages 44–51, Los Alamitos, CA, USA, 2021. IEEE Computer Society.
- [14] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Designing and constructing internet-of-Things systems: An overview of the ecosystem. *Internet of Things*, 19:100529, 2022.
- [15] João Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. A Pattern-Language for Self-Healing Internet-of-Things Systems. In *Proceedings of the European Conference on Pattern Languages of Programs 2020, EuroPLoP '20*, pages 1–17, New York, NY, USA, 2020. Association for Computing Machinery.
- [16] Miguel Duarte, João Pedro Dias, Hugo Sereno Ferreira, and André Restivo. Evaluation of IoT Self-Healing Mechanisms Using Fault-Injection in Message Brokers. In *Proceedings of the 4th International Workshop on Software Engineering Research and Practice for the IoT, SERP4IoT '22*, page 9–16, New York, NY, USA, 2022. Association for Computing Machinery.
- [17] Farshad Firouzi, Bahar Farahani, Markus Weinberger, Gabriel DePace, and Fereidoon Shams Aliee. Iot fundamentals: Definitions, architectures, challenges, and promises. *Intelligent Internet of Things: From Device to Fog and Cloud*, pages 3–50, 2020.
- [18] Ye Fu, Daojuan Zhang, Chonghua Wang, Xi Luo, Wenxin Liu, Tianjie Luo, and Fuyang Fang. Conflict Detection of Device Linkage Rules in Smart Home Systems. *2020 Chinese Automation Congress (CAC)*, pages 3160–3166, 2020.
- [19] GEXF Working Group. GEXF Format. <https://gexf.net/>, 2009. [Online; Accessed 2023].
- [20] Bing Huang, Hai Dong, and Athman Bouguettaya. Conflict Detection in IoT-based Smart Homes. *2021 IEEE International Conference on Web Services (ICWS)*, pages 303–313, 2021.
- [21] IBM. IBM IoT Definition. <https://www.ibm.com/blogs/internet-of-things/what-is-the-iot/>. [Online; Accessed 2023].
- [22] Abhishek Khanna and Sanmeet Kaur. Internet of Things (IoT), Applications and Challenges: A Comprehensive Review. *Wireless Personal Communications*, 114(2):1687–1762, 2020.
- [23] André Sousa Lago, João Pedro Dias, and Hugo Sereno Ferreira. Conversational Interface for Managing Non-trivial Internet-of-Things Systems. In *Computational Science – ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part V*, pages 384–397, Berlin, Heidelberg, 2020. Springer-Verlag.
- [24] André Sousa Lago, João Pedro Dias, and Hugo Sereno Ferreira. Managing non-trivial internet-of-things systems with conversational assistants: A prototype and a feasibility experiment. *Journal of Computational Science*, 51:101324, 2021.

- [25] Ya-Hua Lee and Fuchun Joseph Lin. Situation Awareness and Conflict Resolution in Smart Home with Multiple Users. *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 852–857, 2019.
- [26] Xinyi Li, Lei Zhang, and Xipeng Shen. IA-Graph Based Inter-App Conflicts Detection in Open IoT Systems. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019*, page 135–147, New York, NY, USA, 2019. Association for Computing Machinery.
- [27] Meiyi Ma, Sarah Masud Preum, and John A. Stankovic. CityGuard: A Watchdog for Safety-Aware Conflict Detection in Smart Cities. *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 259–270, 2017.
- [28] Marco Manca, Fabio Paternò, Carmen Santoro, and Luca Corcella. Supporting end-user debugging of trigger-action rules for IoT applications. *International Journal of Human-Computer Studies*, 123:56–69, 2019.
- [29] Isac Daniel de Figueiredo Novo and Pedro Ribeiro. *On the Summarization of Complex Networks*. Master’s Thesis, Faculdade de Ciências da Universidade do Porto, Portugal, 2022.
- [30] Oracle. Oracle IoT Definition. <https://www.oracle.com/pt/internet-of-things/what-is-iot/>. [Online; Accessed 2023].
- [31] Pavana Pradeep and Krishna Kant. Conflict Detection and Resolution in IoT Systems: A Survey. *IoT*, 3(1):191–218, 2022.
- [32] Sílvia Resendes, Paulo Carreira, and André C. Santos. Conflict detection and resolution in home and building automation systems: a literature review. *Journal of Ambient Intelligence and Humanized Computing*, 5(5):699–715, 2014.
- [33] André Restivo and Ademar Aguiar. Towards Detecting and Solving Aspect Conflicts and Interferences Using Unit Tests. In *Proceedings of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies, SPLAT ’07*, page 7–es, New York, NY, USA, 2007. Association for Computing Machinery.
- [34] André Restivo and Ademar Aguiar. Disciplined Composition of Aspects Using Tests. In *Proceedings of the 2008 AOSD Workshop on Linking Aspect Technology and Evolution, LATE ’08*, pages 1–5, New York, NY, USA, 2008. Association for Computing Machinery.
- [35] Trusit Shah, S. Venkatesan, Tiffany Ngo, Pratima, and Karthikeyan Neelamegam. Conflict Detection in Rule Based IoT Systems. *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 0276–0284, 2019.
- [36] Margarida Silva, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. A Review on Visual Programming for Distributed Computation in IoT. In *Computational Science – ICCS 2021: 21st International Conference, Krakow, Poland, June 16-18, 2021, Proceedings, Part IV*, pages 443–457, Cham, 2021. Springer International Publishing.
- [37] Thais R. M. Braga Silva, Linnyer Ruiz, and Antonio A. F. Loureiro. How to Conciliate Conflicting Users’ Interests for Different Collective, Ubiquitous and Context-aware Applications? In *IEEE Local Computer Network Conference*, pages 288–291, 2010.

- [38] Danny Soares, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Programming IoT-Spaces: A User-Survey on Home Automation Rules. In *Computational Science – ICCS 2021: 21st International Conference, Krakow, Poland, June 16-18, 2021, Proceedings, Part IV*, pages 512–525, 2021.
- [39] Yan Sun, Xukai Wang, Hong Luo, and Xiangyang Li. Conflict Detection Scheme Based on Formal Rule Model for Smart Building Systems. *IEEE Transactions on Human-Machine Systems*, 45(2):215–227, 2015.
- [40] Emmanuel Munguia Tapia, Stephen S. Intille, and Kent Larson. Activity recognition in the home using simple and ubiquitous sensors. In *Pervasive Computing*, pages 158–175, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [41] Kenneth W Thomas. Conflict and conflict management: Reflections and update. *Journal of organizational behavior*, pages 265–274, 1992.
- [42] Diogo Torres, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Real-Time Feedback in Node-RED for IoT Development: An Empirical Study. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–8. IEEE, 2020.
- [43] Wikipedia. Components (graph theory). [https://en.wikipedia.org/wiki/Component_\(graph_theory\)](https://en.wikipedia.org/wiki/Component_(graph_theory)), 2023. [Online; Accessed 2023].
- [44] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14*, New York, NY, USA, 2014. Association for Computing Machinery.
- [45] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenburg, Shan Lu, and Blase Ur. Autotap: Synthesizing and repairing trigger-action programs using ltl properties. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 281–291, 2019.

Appendix A

dfs() Function Implementation

This chapter includes the code we build to implement *dfs()* function.

```
1 dfs(vertex, visited, stack, loops, visitedPaths) {
2   const vertexId = vertex.getId()
3   visited.add(vertexId)
4   stack.add(vertexId)
5
6   for (const edge of vertex.getEdges()) {
7     const endVertex = edge.getEndVertex()
8     const endVertexId = endVertex.getId()
9
10    if (stack.has(endVertexId)) {
11      const loopPath = Array.from(stack).slice(
12        Array.from(stack).indexOf(endVertexId)
13      )
14      loopPath.push(endVertexId)
15      let loopPathC = [...loopPath]
16      // console.log(loopPath)
17      if (this.checkNewPath(visitedPaths, loopPathC)) {
18        if (loopPath.length > 3) {
19          loops.push({
20            vertexIds: loopPath,
21            associatedRules: loopPath.map((id, index) => {
22              if (index === loopPath.length - 1) {
23                return null
24              }
25              const nextId = loopPath[index + 1]
26              const vertex = this.getVertex(id)
27              const edge = vertex
28                .getEdges()
29                .find(
30                  (e) =>
```

```
31         e.getEndVertex().getId() === nextId
32     )
33     return edge.getAssociatedRule()
34 },
35 })
36 } else if (loopPath.length === 3) {
37     const combinations =
38         this.generateRuleCombinations(loopPath)
39
40     for (let combination of combinations) {
41         loops.push({
42             vertexIds: loopPath,
43             associatedRules: combination,
44         })
45     }
46 }
47 visitedPaths.push(loopPath)
48 }
49 }
50
51 if (!visited.has(endVertexId)) {
52     this.dfs(endVertex, visited, stack, loops, visitedPaths)
53 }
54 }
55
56 stack.delete(vertexId)
57 visited.delete(vertexId) // Remove vertex from visited set to allow revisiting
58     it in different paths
59 }
```

Listing A.1: dfs() function

Appendix B

List of Smart Devices

This annex includes the list of smart devices in our designed smart environment. It is divided into divisions, and for each one, devices are differentiated between detectors/sensors and actuators existing there. Next to each device are the states it can be found in or the range of values it works.

Common:

Detectors: (detectors that exist in every division)

- Motion. -> on/off
- Smoke. -> on/off
- Air quality sensors. -> on/off

Actuators: (actuators not associated with a specific division)

- Hot water system. -> on/off
- Robot vacuum cleaner. -> on/off

Garage:

Detectors:

- Garage door sensor. -> open/close
- Lights sensor. -> on/off
- Washing machine sensor. -> on/off
- Dryer sensor. -> on/off
- Inside door sensor. -> open/close

Actuators:

- Garage door. -> open/close

- Inside door. -> open/close
- Washing machine. -> on/off
- Dryer. -> on/off
- Lights. -> off, 0% - 100% (brightness)

Patio:

Detectors:

- Entry door sensor. -> open/close
- Lights sensor. -> on/off

Actuators:

- Entry door. -> open/close
- Lights. -> off, 0% - 100% (brightness)

Pool:

Detectors:

- Pool cover sensor. -> open/close
- Water temperature sensor. -> off, 5°C - 35°C
- Cleaning system sensor. -> on/off

Actuators:

- Pool cover. -> open/close
- Cleaning system. -> on/off
- Water heating system. -> off, 18°C - 30°C (temperature)

Garden:

Detectors:

- Humidity sensor. -> off, 0% - 100%
- Temperature sensor. -> off, -15°C - 45°C
- Lights sensor. -> on/off
- Watering system sensor. -> on/off
- Robot lawn mower sensor. -> on/off

Actuators:

- Watering system. -> on/off
- Robot lawn mower. -> on/off
- Lights. -> off, 0% - 100% (brightness)

Living Room:

Detectors:

- Temperature sensor. -> off, -15°C - 45°C
- Windows sensor. -> open/close
- Blinds sensor. -> open/close
- Lights sensor. -> on/off
- TV sensor. -> on/off
- HVAC sensor. -> on/off
- Sound system sensor. -> on/off

Actuators:

- TV. -> on/off
- HVAC. -> off, 16°C - 30°C (temperature)
- Sound system. -> on/off
- Windows. -> open/close
- Blinds. -> open/close
- Lights. -> off, 0% - 100% (brightness)

Kitchen:

Detectors:

- Lights sensor. -> on/off
- Stove sensor. -> on/off
- Oven sensor. -> on/off
- Hood sensor. -> on/off
- Dishwasher sensor. -> on/off
- Coffee machine sensor. -> on/off
- Fryer sensor. -> on/off
- TV sensor. -> on/off

Actuators:

- Stove. -> on/off
- Oven. -> on/off
- Hood. -> on/off
- Dishwasher. -> on/off
- Coffee machine. -> on/off
- Fryer. -> on/off
- TV. -> on/off
- Lights. -> off, 0% - 100% (brightness)

Bedroom:**Detectors:**

- Temperature sensor. -> off, -15°C - 45°C
- Windows sensor. -> open/close
- Blinds sensor. -> open/close
- Lights sensor. -> on/off
- TV sensor. -> on/off
- Bedside lamps sensor. -> off, 0% - 100% (brightness)
- HVAC sensor. -> on/off

Actuators:

- TV. -> on/off
- Bedside lamps. -> off, 0% - 100% (brightness)
- HVAC. -> off, 16°C - 30°C (temperature)
- Windows. -> open/close
- Blinds. -> open/close
- Lights. -> off, 0% - 100% (brightness)

Bathroom:**Detectors:**

- Lights sensor. -> on/off
- Heated towel rack sensor. -> on/off

Actuators:

- Heated towel rack. -> on/off
- Lights. -> off, 0% - 100% (brightness)

Appendix C

Query to Generate Initial Rule Set

This annex includes the query we introduced in ChatGPT to generate the initial rules with no conflicts between them. In it, we describe the existing divisions in the smart home and provide the list of devices existing in it, as well as some considerations that must be taken into account for the generation of rules. Lastly, we present some examples to demonstrate more concretely what we intend.

Consider a smart home with 1 living room, 1 kitchen, 1 bedroom, 1 garage, 1 bathroom, 1 garden, 1 pool, and 1 patio. The list of smart devices that exist in this house is provided then and is divided into divisions. The sensors and actuators that are inside the 'Common' section are devices that are common to every division (there is no common division in the house). The smart devices list is the following (next to each device are the possible states or range of values in which each works):

Common:

Detectors: (detectors that exist in every division)

- Motion. -> on/off
- Smoke. -> on/off
- Air quality sensors. -> on/off

Actuators: (actuators not associated with a specific division)

- Hot water system. -> on/off
- Robot vacuum cleaner. -> on/off

Garage:

Detectors:

- Garage door sensor. -> open/close
- Lights sensor. -> on/off
- Washing machine sensor. -> on/off
- Dryer sensor. -> on/off
- Inside door sensor. -> open/close

Actuators:

- Garage door. -> open/close
- Inside door. -> open/close
- Washing machine. -> on/off
- Dryer. -> on/off
- Lights. -> off, 0% - 100% (brightness)

Patio:

Detectors:

- Entry door sensor. -> open/close
- Lights sensor. -> on/off

Actuators:

- Entry door. -> open/close
- Lights. -> off, 0% - 100% (brightness)

Pool:

Detectors:

- Pool cover sensor. -> open/close
- Water temperature sensor. -> off, 5°C - 35°C
- Cleaning system sensor. -> on/off

Actuators:

- Pool cover. -> open/close
- Cleaning system. -> on/off
- Water heating system. -> off, 18°C - 30°C (temperature)

Garden:

Detectors:

- Humidity sensor. -> off, 0% - 100%
- Temperature sensor. -> off, -15°C - 45°C
- Lights sensor. -> on/off
- Watering system sensor. -> on/off
- Robot lawn mower sensor. -> on/off

Actuators:

- Watering system. -> on/off
- Robot lawn mower. -> on/off
- Lights. -> off, 0% - 100% (brightness)

Living Room:

Detectors:

- Temperature sensor. -> off, -15°C - 45°C
- Windows sensor. -> open/close
- Blinds sensor. -> open/close
- Lights sensor. -> on/off
- TV sensor. -> on/off
- HVAC sensor. -> on/off
- Sound system sensor. -> on/off

Actuators:

- TV. -> on/off
- HVAC. -> off, 16°C - 30°C (temperature)
- Sound system. -> on/off
- Windows. -> open/close
- Blinds. -> open/close
- Lights. -> off, 0% - 100% (brightness)

Kitchen:

Detectors:

- Lights sensor. -> on/off
- Stove sensor. -> on/off
- Oven sensor. -> on/off

- Hood sensor. -> on/off
- Dishwasher sensor. -> on/off
- Coffee machine sensor. -> on/off
- Fryer sensor. -> on/off
- TV sensor. -> on/off

Actuators:

- Stove. -> on/off
- Oven. -> on/off
- Hood. -> on/off
- Dishwasher. -> on/off
- Coffee machine. -> on/off
- Fryer. -> on/off
- TV. -> on/off
- Lights. -> off, 0% - 100% (brightness)

Bedroom:**Detectors:**

- Temperature sensor. -> off, -15°C - 45°C
- Windows sensor. -> open/close
- Blinds sensor. -> open/close
- Lights sensor. -> on/off
- TV sensor. -> on/off
- Bedside lamps sensor. -> off, 0% - 100% (brightness)
- HVAC sensor. -> on/off

Actuators:

- TV. -> on/off
- Bedside lamps. -> off, 0% - 100% (brightness)
- HVAC. -> off, 16°C - 30°C (temperature)
- Windows. -> open/close
- Blinds. -> open/close
- Lights. -> off, 0% - 100% (brightness)

Bathroom:

Detectors:

- Lights sensor. -> on/off
- Heated towel rack sensor. -> on/off

Actuators:

- Heated towel rack. -> on/off
- Lights. -> off, 0% - 100% (brightness)

Some considerations to take into account:

- Rules must have the periodicity with which they must be executed (like always, every Thursday at 10 PM, every day at 20:30, ...). You can use minutes, hours, days of the month, months, and days of the week for that periodicity;
- When turning on the HVAC, a temperature must be specified;
- When the humidity sensor is used, a percentage must be specified;
- When using the temperature sensors, a temperature must be specified;
- When turning on the water heating system, a temperature must be specified;
- When turning on the lights, the desired brightness percentage must be specified;
- It's not possible to use time intervals in triggers (like 'If the garage door is opened for more than 10 minutes');
- Each rule can have many triggers separated by 'and', but only one action that should influence only one specific device (like 'living room HVAC', not 'living room HVAC and bedroom HVAC', or 'turn on living room lights and the TV');
- Some rules might not have a trigger part, only an action (eg, 'Turn on the living room lights every day at 10 PM');
- Weather states, like raining, sunny, windy, etc, are not allowed;
- In triggers, the only devices that can be used are the detectors, while the actuators can only be used in the action part.

Here are some examples to easily illustrate the type of rules intended and their format:

1. If the outside temperature is below 10°C and the living room door is closed, set the living room HVAC temperature to 22°C. (Daily from 6 PM to 11:59 PM)
2. If motion is detected in the bedroom and lights are off, set bedroom lights brightness to 35%. (Daily at 8 PM)

3. Turn on the pool cleaning system. (Every Wednesday at 10 PM)

Taking into account these devices, write a set of 35 rules in IFTTT format without having conflicts between them. The set of generated rules must be diversified, trying to use different devices and periodicities.

Appendix D

Initial Set of Rules

This chapter includes the initial set of rules initially generated for the smart environment we designed.

1. If motion is detected in the living room and living room lights are off, set the living room lights brightness to 50%. (Always)
2. If smoke is detected in the living room, open the windows in the living room. (Always)
3. If the air quality in the living room is poor, open the windows in the living room. (Always)
4. When the garage door is open, set the garage lights brightness to 35%. (Always)
5. If the patio entry door is open, set the patio lights brightness to 75%. (Always)
6. If the living room temperature exceeds 28°C, set the living room HVAC to 22°C. (Always)
7. If the bedroom temperature falls below 18°C, set the bedroom HVAC to 20°C. (Always)
8. If the pool cover is closed and the water temperature reaches 25°C, open the pool cover. (Always)
9. Set the garage lights brightness to 35% every day at 8 PM. (Daily at 8 PM)
10. If the humidity in the garden is below 30%, turn on the garden watering system. (Always)
11. If the temperature in the garden is below 10°C and garden lights are off, turn on the garden lights and set the brightness to 20%. (Always)
12. If the living room TV is off, turn on the living room TV. (Always)
13. When the kitchen lights are on, turn on the kitchen TV. (Always)
14. Turn off the bedroom HVAC system every day at 11 PM. (Daily at 11 PM)

15. Set the garden lights brightness to 20% every day at 7 PM. (Daily at 7 PM)
16. Turn on the hot water system every day at 6 AM. (Daily at 6 AM)
17. If the pool cleaning system is off, turn it on every Wednesday at 10 AM. (Weekly on Wednesdays at 10 AM)
18. If the garage lights are off and the garage door is open, set the garage lights brightness to 35%. (Always)
19. If the patio lights are off and the patio entry door is open, set the patio lights brightness to 75%. (Always)
20. If the living room lights are off and the living room door is open, set the living room lights brightness to 50%. (Always)
21. Set the patio lights brightness to 75% every day at 9 PM. (Daily at 9 PM)
22. Set the bedroom bedside lamps brightness to 20% every day at 8:30 PM. (Daily at 8:30 PM)
23. Set the kitchen lights brightness to 45% every day at 7 AM. (Daily at 7 PM)
24. Set the bathroom lights brightness to 28% every day at 6:30 AM. (Daily at 6:30 PM)
25. Turn on the garden watering system every Monday, Wednesday, and Friday at 6 AM. (Every Monday, Wednesday and Friday at 6 AM)
26. Set the garage lights brightness to 35% every Tuesday and Thursday at 7 PM. (Every Tuesday and Thursday at 7 PM)
27. Turn on the living room TV every day at 8 PM. (Daily at 8 PM)
28. Turn on the pool cleaning system every Saturday at 9 AM. (Every Saturday at 9 AM)
29. Turn on the dishwasher in the kitchen every day at 10 AM. (Daily at 10 AM)
30. Turn on the heated towel rack in the bathroom every day at 7:30 AM. (Daily at 7:30 AM)
31. If the living room temperature falls below 15°C, set the living room HVAC to 23°C. (Always)
32. If the bedroom temperature exceeds 25°C, set the bedroom HVAC to 22°C. (Always)
33. If the outside temperature is below 15°C and the pool water temperature is below 20°C, set the pool water heating system temperature to 25°C. (Always)
34. If the outside temperature exceeds 30°C and the pool water temperature is above 23°C, turn off the pool water heating system. (Always)
35. If the bedroom temperature falls below 10°C, set the bedroom HVAC to 23°C. (Daily from 0 AM to 7 AM)

Appendix E

Updated Rule Set

This chapter includes the updated set of rules generated for the smart environment we designed after some conflicts were detected in Section 6.2.1.1 (p. 62).

1. If motion is detected in the living room and living room lights are off, set the living room lights brightness to 50%. (Always)
2. If smoke is detected in the living room, open the windows in the living room. (Always)
3. If the air quality in the living room is poor, open the windows in the living room. (Always)
4. When the garage door is open, set the garage lights brightness to 35%. (Always)
5. If the patio entry door is open, set the patio lights brightness to 75%. (Always)
6. If the living room temperature exceeds 28°C, set the living room HVAC to 22°C. (Always)
7. If the bedroom temperature falls below 18°C, set the bedroom HVAC to 20°C. (Daily from 4 PM to 10:59 PM)
8. If the pool cover is closed and the water temperature reaches 25°C, open the pool cover. (Always)
9. Set the garage lights brightness to 35% every day at 8 PM. (Daily at 8 PM)
10. If the humidity in the garden is below 30%, turn on the garden watering system. (Always)
11. If the temperature in the garden is below 10°C and garden lights are off, turn on the garden lights and set the brightness to 20%. (Always)
12. If the living room TV is off, turn on the living room TV. (Always)
13. When the kitchen lights are on, turn on the kitchen TV. (Always)

14. Turn off the bedroom HVAC system every day at 11 PM. (Daily at 11 PM)
15. Set the garden lights brightness to 20% every day at 7 PM. (Daily at 7 PM)
16. Turn on the hot water system every day at 6 AM. (Daily at 6 AM)
17. If the pool cleaning system is off, turn it on every Wednesday at 10 AM. (Weekly on Wednesdays at 10 AM)
18. If the garage lights are off and the garage door is open, set the garage lights brightness to 35%. (Always)
19. If the patio lights are off and the patio entry door is open, set the patio lights brightness to 75%. (Always)
20. If the living room lights are off and the living room door is open, set the living room lights brightness to 50%. (Always)
21. Set the patio lights brightness to 75% every day at 9 PM. (Daily at 9 PM)
22. Set the bedroom bedside lamps brightness to 20% every day at 8:30 PM. (Daily at 8:30 PM)
23. Set the kitchen lights brightness to 45% every day at 7 AM. (Daily at 7 AM)
24. Set the bathroom lights brightness to 28% every day at 6:30 AM. (Daily at 6:30 AM)
25. Turn on the garden watering system every Monday, Wednesday, and Friday at 6 AM. (Every Monday, Wednesday and Friday at 6 AM)
26. Set the garage lights brightness to 35% every Tuesday and Thursday at 7 PM. (Every Tuesday and Thursday at 7 PM)
27. Turn on the living room TV every day at 8 PM. (Daily at 7 PM)
28. Turn on the pool cleaning system every Saturday at 9 AM. (Every Saturday at 9 AM)
29. Turn on the dishwasher in the kitchen every day at 10 AM. (Daily at 10 AM)
30. Turn on the heated towel rack in the bathroom every day at 7:30 AM. (Daily at 7:30 AM)
31. If the living room temperature falls below 15°C, set the living room HVAC to 23°C. (Always)
32. If the bedroom temperature exceeds 25°C, set the bedroom HVAC to 22°C. (Daily from 4 PM to 10:59 PM)
33. If the outside temperature is below 15°C and the pool water temperature is below 20°C, set the pool water heating system temperature to 25°C. (Always)

34. If the outside temperature exceeds 30°C and the pool water temperature is above 23°C, turn off the pool water heating system. (Always)
35. If the bedroom temperature falls below 10°C, set the bedroom HVAC to 23°C. (Daily from 0 AM to 7 AM)

Appendix F

Redundant Rules Generated

This chapter includes the redundant rules generated for the smart environment we designed.

36. If the living room temperature exceeds 25°C, set the living room HVAC to 22°C. (Always)
37. If the living room temperature exceeds 30°C, set the living room HVAC to 22°C. (Every Saturday and Sunday between June and September from 10 AM to 6 PM)
38. If the living room temperature falls below 20°C, set the living room HVAC to 23°C. (Daily in November between 7 PM and 10 PM)
39. If the bedroom temperature exceeds 22°C, set the bedroom HVAC to 22°C. (Daily from 3 PM to 10:59 PM)
40. If the bedroom temperature falls below 15°C, set the bedroom HVAC to 20°C. (Daily from 5 PM to 8:59 PM)
41. If the outside temperature exceeds 30°C and the pool water temperature is above 25°C, turn off the pool water heating system. (Always)
42. If the outside temperature is below 10°C and the pool water temperature is below 20°C, turn on the pool water heating system and set the target temperature to 25°C. (Always)
43. If the outside temperature exceeds 35°C and the pool water temperature is above 24°C, turn off the pool water heating system. (Daily between May and September)
44. If the outside temperature is below 5°C and the pool water temperature is below 18°C, set the pool water heating system temperature to 25°C. (Every Friday, Saturday and Sunday in December and January from 10 AM to 7 PM)

Appendix G

Inconsistent Rules Generated

This chapter includes the inconsistent rules generated for the smart environment we designed, specifically, those with the same trigger but different actions on the same actuator as the initially generated rules.

45. If motion is detected in the living room and living room lights are off, set the living room lights brightness to 70%. (Daily at 5 PM)
46. If motion is detected in the living room, turn off the living room lights. (Daily from 8 AM to 6 PM)
47. When the garage door is open, set the garage lights brightness to 80%. (Always)
48. If the patio entry door is open and patio lights are off, set the patio lights brightness to 30%. (Daily from 6 PM to 6 AM)
49. If the living room temperature exceeds 28°C, set the living room HVAC to 20°C. (Always)
50. If the bedroom temperature is below or equal to 18°C, set the bedroom HVAC to 23°C. (Daily from 4 PM to 10:59 PM)
51. Set the garage lights brightness to 10% every day at 8 PM. (Daily at 8 PM)
52. If the humidity in the garden is below 30%, turn off the garden watering system. (Always)
53. If the living room door is open, set the living room lights brightness to 63%. (Daily from 7 PM to 23:59 PM)
54. Set the patio lights brightness to 90% every day at 9 PM. (Daily at 9 PM)
55. Set the bedroom bedside lamps brightness to 50% every day at 8:30 PM. (Daily at 8:30)
56. If the living room temperature exceeds 30°C, set the living room HVAC to 19°C. (Always)

57. If the living room temperature exceeds 25°C, set the living room HVAC to 21°C. (Daily between June and September from 9 AM to 7 PM)
58. If the garage lights are off and the garage gate is closed, set the garage lights brightness to 57%. (Daily between November and March from 6 PM to 10 PM)

Appendix H

Loop Sets Generated

This chapter includes the loop sets generated for the smart environment we designed.

Set 1

- 59. If motion is detected in the living room, set the living room lights brightness to 25%. (Always)
- 60. If the living room lights are off and the living room door is open, set the living room lights brightness to 70%. (Always)
- 61. If motion is detected in the living room, turn off the living room lights. (Always)

Set 2

- 4. When the garage door is open, set the garage lights brightness to 35%. (Always)
- 62. If the garage lights are off and the garage door is open, set the garage lights brightness to 80%. (Always)
- 63. When the garage door is open, turn off the garage lights. (Always)

Set 3

- 1. If motion is detected in the living room and living room lights are off, set the living room lights brightness to 50%. (Always)
- 59. If motion is detected in the living room, set the living room lights brightness to 25%. (Always)
- 61. If motion is detected in the living room, turn off the living room lights. (Always)

Set 4

- 64. If the living room lights are turned on, close the living room blinds. (Daily at 8 PM)
- 65. If the living room blinds are closed, set the living room HVAC to 22°C. (Daily between December and February from 6 PM to 12 PM)
- 66. If the living room HVAC is turned on, set the living room lights brightness to 50%. (Always)

Appendix I

Final Set of Rules

This chapter includes the final set of rules generated by ChatGPT for the smart environment we designed.

1. If motion is detected in the living room and living room lights are off, set the living room lights brightness to 50%. (Always)
2. If smoke is detected in the living room, open the windows in the living room. (Always)
3. If the air quality in the living room is poor, open the windows in the living room. (Always)
4. When the garage door is open, set the garage lights brightness to 35%. (Always)
5. If the patio entry door is open, set the patio lights brightness to 75%. (Always)
6. If the living room temperature exceeds 28°C, set the living room HVAC to 22°C. (Always)
7. If the bedroom temperature falls below 18°C, set the bedroom HVAC to 20°C. (Daily from 4 PM to 10:59 PM)
8. If the pool cover is closed and the water temperature reaches 25°C, open the pool cover. (Always)
9. Set the garage lights brightness to 35% every day at 8 PM. (Daily at 8 PM)
10. If the humidity in the garden is below 30%, turn on the garden watering system. (Always)
11. If the temperature in the garden is below 10°C and garden lights are off, turn on the garden lights and set the brightness to 20%. (Always)
12. If the living room TV is off, turn on the living room TV. (Always)
13. When the kitchen lights are on, turn on the kitchen TV. (Always)

14. Turn off the bedroom HVAC system every day at 11 PM. (Daily at 11 PM)
15. Set the garden lights brightness to 20% every day at 7 PM. (Daily at 7 PM)
16. Turn on the hot water system every day at 6 AM. (Daily at 6 AM)
17. If the pool cleaning system is off, turn it on every Wednesday at 10 AM. (Weekly on Wednesdays at 10 AM)
18. If the garage lights are off and the garage door is open, set the garage lights brightness to 35%. (Always)
19. If the patio lights are off and the patio entry door is open, set the patio lights brightness to 75%. (Always)
20. If the living room lights are off and the living room door is open, set the living room lights brightness to 50%. (Always)
21. Set the patio lights brightness to 75% every day at 9 PM. (Daily at 9 PM)
22. Set the bedroom bedside lamps brightness to 20% every day at 8:30 PM. (Daily at 8:30 PM)
23. Set the kitchen lights brightness to 45% every day at 7 AM. (Daily at 7 AM)
24. Set the bathroom lights brightness to 28% every day at 6:30 AM. (Daily at 6:30 AM)
25. Turn on the garden watering system every Monday, Wednesday, and Friday at 6 AM. (Every Monday, Wednesday and Friday at 6 AM)
26. Set the garage lights brightness to 35% every Tuesday and Thursday at 7 PM. (Every Tuesday and Thursday at 7 PM)
27. Turn on the living room TV every day at 8 PM. (Daily at 7 PM)
28. Turn on the pool cleaning system every Saturday at 9 AM. (Every Saturday at 9 AM)
29. Turn on the dishwasher in the kitchen every day at 10 AM. (Daily at 10 AM)
30. Turn on the heated towel rack in the bathroom every day at 7:30 AM. (Daily at 7:30 AM)
31. If the living room temperature falls below 15°C, set the living room HVAC to 23°C. (Always)
32. If the bedroom temperature exceeds 25°C, set the bedroom HVAC to 22°C. (Daily from 4 PM to 10:59 PM)
33. If the outside temperature is below 15°C and the pool water temperature is below 20°C, set the pool water heating system temperature to 25°C. (Always)

34. If the outside temperature exceeds 30°C and the pool water temperature is above 23°C, turn off the pool water heating system. (Always)
35. If the bedroom temperature falls below 10°C, set the bedroom HVAC to 23°C. (Daily from 0 AM to 7 AM)
36. If the living room temperature exceeds 25°C, set the living room HVAC to 22°C. (Always)
37. If the living room temperature exceeds 30°C, set the living room HVAC to 22°C. (Every Saturday and Sunday between June and September from 10 AM to 6 PM)
38. If the living room temperature falls below 20°C, set the living room HVAC to 23°C. (Daily in November between 7 PM and 10 PM)
39. If the bedroom temperature exceeds 22°C, set the bedroom HVAC to 22°C. (Daily from 3 PM to 10:59 PM)
40. If the bedroom temperature falls below 15°C, set the bedroom HVAC to 20°C. (Daily from 5 PM to 8:59 PM)
41. If the outside temperature exceeds 30°C and the pool water temperature is above 25°C, turn off the pool water heating system. (Always)
42. If the outside temperature is below 10°C and the pool water temperature is below 20°C, turn on the pool water heating system and set the target temperature to 25°C. (Always)
43. If the outside temperature exceeds 35°C and the pool water temperature is above 24°C, turn off the pool water heating system. (Daily between May and September)
44. If the outside temperature is below 5°C and the pool water temperature is below 18°C, set the pool water heating system temperature to 25°C. (Every Friday, Saturday and Sunday in December and January from 10 AM to 7 PM)
45. If motion is detected in the living room and living room lights are off, set the living room lights brightness to 70%. (Daily at 5 PM)
46. If motion is detected in the living room, turn off the living room lights. (Daily from 8 AM to 6 PM)
47. When the garage door is open, set the garage lights brightness to 80%. (Always)
48. If the patio entry door is open and patio lights are off, set the patio lights brightness to 30%. (Daily from 6 PM to 6 AM)
49. If the living room temperature exceeds 28°C, set the living room HVAC to 20°C. (Always)
50. If the bedroom temperature is below or equal to 18°C, set the bedroom HVAC to 23°C. (Daily from 4 PM to 10:59 PM)

51. Set the garage lights brightness to 10% every day at 8 PM. (Daily at 8 PM)
52. If the humidity in the garden is below 30%, turn off the garden watering system. (Always)
53. If the living room door is open, set the living room lights brightness to 63%. (Daily from 7 PM to 23:59 PM)
54. Set the patio lights brightness to 90% every day at 9 PM. (Daily at 9 PM)
55. Set the bedroom bedside lamps brightness to 50% every day at 8:30 PM. (Daily at 8:30)
56. If the living room temperature exceeds 30°C, set the living room HVAC to 19°C. (Always)
57. If the living room temperature exceeds 25°C, set the living room HVAC to 21°C. (Daily between June and September from 9 AM to 7 PM)
58. If the garage lights are off and the garage gate is closed, set the garage lights brightness to 57%. (Daily between November and March from 6 PM to 10 PM)
59. If motion is detected in the living room, set the living room lights brightness to 25%. (Always)
60. If the living room lights are off and the living room door is open, set the living room lights brightness to 70%. (Always)
61. If motion is detected in the living room, turn off the living room lights. (Always)
62. If the garage lights are off and the garage door is open, set the garage lights brightness to 80%. (Always)
63. When the garage door is open, turn off the garage lights. (Always)

