Chair for Computer Science 10 (Media Computing and Human-Computer Interaction)

RWTH AACHEN UNIVERSITY

# *Visual Trace Debugging of Smart Home Rules*

Master's Thesis
submitted to the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

*by*
*Amit Kumar Shaw*

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Ulrik Schroeder

Registration date: 03.08.2023
Submission date: 05.02.2024

Zentrales Prüfungsamt/Central Examination Office

# Eidesstattliche Versicherung

## Statutory Declaration in Lieu of an Oath

| | |
|---|---|
| ___________________________ | ___________________________ |
| Name, Vorname/Last Name, First Name | Matrikelnummer (freiwillige Angabe) Matriculation No. (optional) |

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

___________________________________________________________________________

___________________________________________________________________________

___________________________________________________________________________

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

| | |
|---|---|
| ___________________________ | ___________________________ |
| Ort, Datum/City, Date | Unterschrift/Signature |

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

**Belehrung:**
**Official Notification:**

**§ 156 StGB: Falsche Versicherung an Eides Statt**

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

**Para. 156 StGB (German Criminal Code): False Statutory Declarations**

Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

**§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt**

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

**Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence**

(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.

(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

I have read and understood the above official notification:

| | |
|---|---|
| ___________________________ | ___________________________ |
| Ort, Datum/City, Date | Unterschrift/Signature |

# Contents

# List of Figures

# List of Tables

# Abstract

Smart homes are revolutionizing our daily lives by using technology to make our routines more convenient and comfortable. With the help of smart homes, people can automate tasks such as turning off lights and adjusting room temperature based on their preferences without manual operation. Trigger-Action Programming (TAP) allows inexperienced users to define automation and interactions between smart home devices without requiring extensive technical knowledge. However, conflicts between TAP rules can lead to unexpected situations, frustrating users and reducing their trust in the system. Researchers have explored two approaches to tackle these challenges and user frustrations: static analysis and dynamic analysis. Although both methods are effective in their ways, they have their limitations and need to provide long-term solutions and debugging support.

This thesis explores the potential of visualizing traces of conflicting TAP rules to identify the root cause of issues in smart homes. Our work builds on previous studies that have shown the effectiveness of visual aids in understanding TAP rules. In this thesis, we developed and evaluated a visual debugging dashboard that traces events associated with activated TAP rules in smart homes and shows the causal relationship between events. We evaluated two dashboard versions - *Baseline*, which traces all activated rules without data filtering, and *Filter*, which incorporates data tracing and filtering. A within-group user study with 16 participants assessed different TAP debugging aspects. Although the *Filter* version received higher System Causability Scale ratings, no significant differences in debugging performance were observed compared to *Baseline*. The results discovered significant differences in outcomes based on bug types. This suggests that different bugs require varied cognitive abilities for effective TAP conflict resolution.

# Überblick

Smart Homes revolutionieren unser tägliches Leben, indem sie mithilfe von Technologie unsere Abläufe bequemer und komfortabler machen. Mit Hilfe von Smart Homes können Menschen Aufgaben wie das Ausschalten des Lichts und die Anpassung der Raumtemperatur nach ihren Wünschen ohne manuelle Bedienung automatisieren. Mithilfe der Trigger-Action-Programming (TAP) können unerfahrene Benutzer Automatisierungen und Interaktionen zwischen Smart-Home-Geräten definieren, ohne dass umfangreiche technische Kenntnisse erforderlich sind. Allerdings können Konflikte zwischen TAP-Regeln zu unerwarteten Situationen führen, die Benutzer frustrieren und ihr Vertrauen in das System verringern. Forscher haben zwei Ansätze untersucht, um diese Herausforderungen und Benutzerfrustrationen anzugehen: statische Analyse und dynamische Analyse. Obwohl beide Methoden auf ihre Art effektiv sind, haben sie ihre Grenzen und müssen langfristige Lösungen und Debugging-Unterstützung bieten.

Diese Arbeit untersucht das Potenzial der Visualisierung widersprüchlicher TAP-Regeln, um die Grundursache von Problemen in Smart Homes zu identifizieren. Unsere Arbeit baut auf früheren Studien auf, die die Wirksamkeit visueller Hilfsmittel beim Verständnis von TAP-Regeln gezeigt haben. In dieser Arbeit haben wir ein visuelles Debugging-Dashboard entwickelt und evaluiert, das Ereignisse im Zusammenhang mit aktivierten TAP-Regeln in Smart Homes nachverfolgt und den kausalen Zusammenhang zwischen Ereignissen aufzeigt. Wir haben zwei Dashboard-Versionen evaluiert: *Baseline*, das alle aktivierten Regeln ohne Datenfilterung verfolgt, und *Filter*, das Datenverfolgung und -filterung umfasst. In einer within-group Benutzerstudie mit 16 Teilnehmern wurden verschiedene TAP-Debugging-Aspekte bewertet. Obwohl die Filterversion höhere System Causability Scale-Bewertungen erhielt, wurden im Vergleich zur *Baseline* keine signifikanten Unterschiede in der Debugging-Leistung beobachtet. Die Ergebnisse zeigten signifikante Unterschiede in den Ergebnissen je nach Fehlertyp. Dies deutet darauf hin, dass verschiedene Fehler unterschiedliche kognitive Fähigkeiten für eine effektive TAP-Konfliktlösung erfordern.

# Acknowledgements

I would like to thank Prof. Dr. Jan Borchers and Prof. Dr. Ulrik Schroeder for examining my thesis.

I would like to thank Adrian Wagner for providing me with dedicated and motivating supervision throughout my work. Your valuable feedback has been instrumental in helping me learn and improve. Thank you so much.

I would like to thank everyone who participated in the studies and took the time to help me with my research. It would not have been possible without their valuable contribution.

Thank you, Werner and Waltraud. I appreciate your valuable inputs and support and the time you took to review my thesis.

Thank you to my family and friends for your endless love, unwavering support, and understanding. You have been my pillars during this journey, and I am grateful for each one of you.

# Conventions

Throughout this thesis, we use the following conventions.

Definitions of technical terms or short excursus are set off in coloured boxes.

**EXCURSUS:**
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
*Excursus*

Named tools and concepts are written in italics.

*Dashboard*

Source code and implementation symbols are written in typewriter-style text.

`SensorLog`

The whole thesis is written in American English.

The first person is written in the plural form. Unidentified third persons are referred to in the plural form.

# Chapter 1

# Introduction

The advent of smart homes is transforming how we live, bringing unmatched convenience and comfort. With this technology, people can easily automate tasks such as turning off lights after leaving a room or adjusting the room temperature from the comfort of their bed. Smart homes significantly benefit independent individuals, assisting those requiring additional care (Marikyan et al. [2019]). In a smart home, one can program devices like doors, lights, and heating systems to adjust based on their preferences without manually operating them (Nikou [2019]). Smart homes are designed to make our lives easier and more comfortable by using technology to take care of everyday tasks and improve our overall quality of life.

Smart home technology automates tasks and improves the quality of life.

The effectiveness of smart home systems depends on how different devices interact and function together smoothly. These interactions are user-defined, often without needing extensive technical knowledge of the underlying systems. Trigger-Action Programming (TAP), an end-user development paradigm (Lieberman et al. [2006]), enables users to define these interactions according to their preferences without requiring significant technical expertise. TAP follows the "IF [trigger] WHILE [conditions] THEN [action]" format, allowing even inexperienced users to create complex behaviors (Ur et al. [2014]). TAP makes smart home automation accessible and flexible, empowering users to customize smart home functionalities to their specific needs

Trigger-Action Programming (TAP) simplifies smart home automation and can be learned easily.

and preferences. This showcases the perfect blend of user-friendliness and technological sophistication of smart home systems.

Conflicts in TAP rules can create undesired situations that lead to user frustration.

Though TAP simplifies smart home automation, it is susceptible to bugs, like any other programming paradigm. These bugs arise from conflicts between multiple TAP rules. Such conflicts can lead to undesired situations where the smart home system either does things it shouldn't or fails to do something when expected. Consequently, this frustrates the users and reduces their trust in the smart home system (Brush et al. [2011], Mennicken and Huang [2012]).

Current research provides limited solutions for identifying the underlying causes of TAP rule conflicts.

Researchers have explored various methods to tackle the challenges and user frustrations caused by conflicting TAP rules. They have either used a static or dynamic analysis approach. Static analysis, as seen in the research by De Russis and Monge Roffarello [2018] and Corno et al. [2019], aims to identify inconsistencies and redundancies during rule creation. While it can catch conflicts early, it may not eliminate issues in daily usage and lacks debugging support after a conflict occurs. Dynamic analysis, as explored by Coppers et al. [2022], predicts potential conflicts during system operation and provides temporary solutions to users. Although these methods are effective in their own ways, they have their limitations and need to provide long-term solutions and debugging support. These approaches highlight the need for a deeper understanding of the root causes of TAP rule conflicts to develop permanent solutions and enhance user confidence in smart home technologies.

Definition: *Trace-Based Debugging*

**TRACE-BASED DEBUGGING:**
Reiss [1993] introduced trace-based debugging by defining a debugger "that incorporates the notion of time to allow users to easily navigate over the execution of their systems."

In this thesis, we explore the potential of debugging through the visualization of traces of conflicting TAP rules to identify the cause of the issue. Our research is inspired by previous studies that have demonstrated the effectiveness of using visual aids to understand TAP rule differences

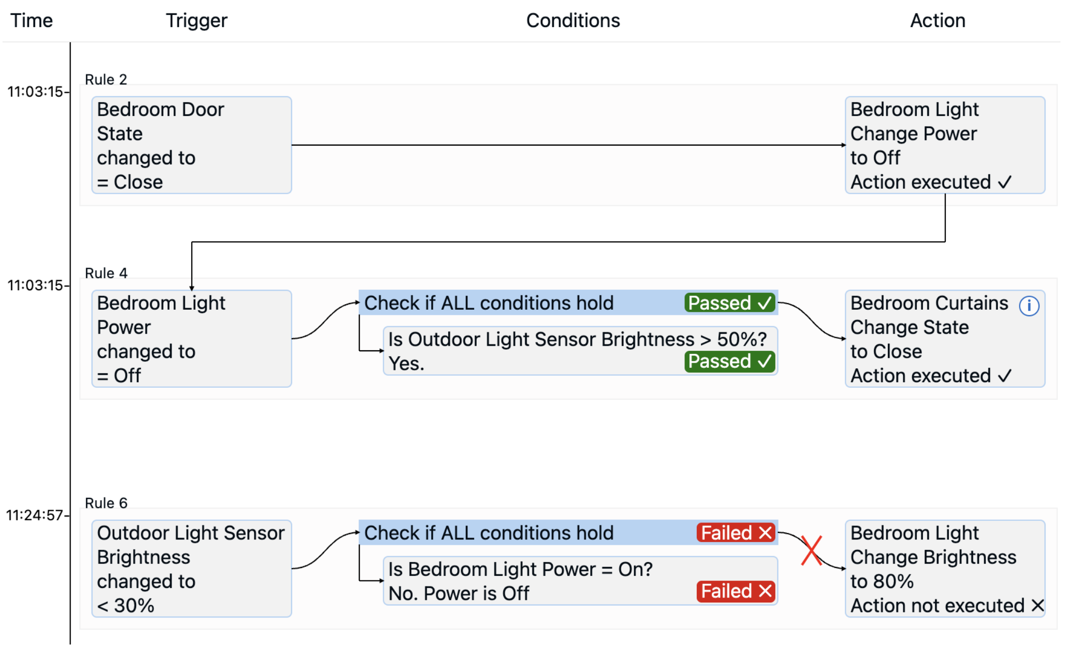

Time
Trigger
Conditions
Action
11:03:15
Rule 2
Bedroom Door State changed to = Close
Bedroom Light Change Power to Off Action executed ✓
11:03:15
Rule 4
Bedroom Light Power changed to = Off
Check if ALL conditions hold Passed ✓
Is Outdoor Light Sensor Brightness > 50%? Yes. Passed ✓
Bedroom Curtains Change State to Close Action executed ✓
11:24:57
Rule 6
Outdoor Light Sensor Brightness changed to < 30%
Check if ALL conditions hold Failed ✕
Is Bedroom Light Power = On? No. Power is Off Failed ✕
Bedroom Light Change Brightness to 80% Action not executed ✕

**Figure 1.1:** The figure shows a part of the *Debugging Dashboard* that represents the events linked with the activated TAP rules in the sequence of their execution.

and interfaces that offer varying levels of detail visualization (De Russis and Corno [2015] and Zhao et al. [2021]). As a result, we have developed and assessed a visual debugging dashboard in our work. Our dashboard, which follows the principles of trace-based debugging (Reiss [1993]), is designed to trace events associated with activated TAP rules in smart homes and display them in the order of execution, as depicted in Figure 1.1. To illustrate the causal relationship between events, it uses arrows, similar to the *Whyline* debugging interface (Figure 2.4), which has shown that visualizing runtime events and representing data and control flow causality can significantly reduce the debugging time (Ko and Myers [2004]). Moreover, the dashboard provides a filtering feature to trace relevant data that leads to activating a rule in a smart home. This feature is intended to help users identify the root cause of conflicts in TAP rules by visualizing data at different levels of detail, thus enhancing their experience with smart home systems.

We investigate the potential of visualizing traces of conflicting TAP rules to identify the cause of the issue in smart homes.

This thesis investigates the importance of data tracing in our visual debugging dashboard. Given the lack of simi-

We evaluate the impact of filtering and tracing relevant data by comparing the *Baseline* and the *Filter* version of the dashboard.

lar existing tools, to our knowledge, we evaluate our dashboard by comparing two versions: the *Baseline* version, which traces all activated TAP rules in a smart home without offering data filtering, and the *Filter* version, which allows for filtering and tracing relevant data. Hence, we address the following questions in our research:

**RQ1** Does filtering and tracing relevant data assist in accurately identifying the cause of conflicts in TAP rules?

- **RQ1.1** Does filtering and tracing relevant data make it easier to identify the cause of conflicts in TAP rules?
- **RQ1.2** Does filtering and tracing relevant data reduce the time taken by users to identify the causes of conflicts in TAP rules?

**RQ2** Does filtering and tracing relevant data impact the user's confidence in identifying the cause of conflicts in TAP rules?

- **RQ2.1** Does filtering and tracing relevant data have an impact on user's understanding of the conflicting situation?

**RQ3** Does filtering and tracing relevant data impact user satisfaction when identifying the cause of conflicts in TAP rules?

This thesis is divided into five chapters.

In the following, we describe the related work in Chapter 2. There, we discuss the development of TAP, different conflicts in TAP rules, and existing solutions using static and dynamic approaches. In Chapter 3, we will explain the debugging dashboard's design choices and implementation details. In Chapter 4, we will discuss the user study conducted to evaluate the dashboard and discuss the study's results. Finally, we conclude in Chapter 5 with a summary and suggestions for future work.

# Chapter 2

# Related Work

This chapter looks at related work in TAP debugging before this thesis. First, we introduce and discuss the development of TAP. Then, we look at different types and definitions of undesired conflicts in TAP rules. Finally, we go through various existing research offering solutions to resolve TAP conflicts using static and dynamic analysis approaches and also look at some visualizations of TAP rules.

## 2.1 Trigger-Action Programming (TAP)

Trigger-Action Programming (TAP) simplifies smart home automation by allowing users to specify automation rules based on a trigger and a predefined action.

Trigger-Action Programming (TAP), an end-user development paradigm (Lieberman et al. [2006]), has evolved as a pivotal concept in the realm of smart home automation, offering a user-friendly way to program smart home environments. This approach allows users to specify automation rules in a format where a specified trigger leads to a predefined action. TAP's structure is simple: "IF [trigger], WHILE [conditions], THEN [action]," for example, "IF light is switched on, WHILE room temperature is above 26°C, THEN turn on the AC." Consequently, TAP is adopted by IFTTT[1], Home Assistant[2], Samsung Smart-

[1]https://ifttt.com (last accessed on February 2, 2024)
[2]https://www.home-assistant.io/ (last accessed on February 2, 2024)

Things[3], Philips Hue[4] and other commercial platforms.

TAP is significant for smart homes and easy for novice users to learn and create rules, as concluded by studies.

Existing research has demonstrated the significance of TAP within smart homes. It has been proven capable of representing the majority of behaviors that potential users may desire (Ghiani et al. [2017], Nacci et al. [2018], Woo and Lim [2015]). According to the study conducted by Ur et al. [2014], it was found that novice users could quickly learn TAP and create rules for expressing smart home behaviors. The researchers developed an interface similar to IFTTT with multiple triggers and actions. Users were asked to mention the smart home behaviors they wanted, and the study found that all behaviors that required programming could be expressed in a trigger-action format. The study concluded that users could correctly create a set of rules about 80% of the time.

Studies revealed that TAP systems have evolved to support complex conditions and user preferences.

Ur et al. [2016] analyzed 224,590 trigger-action programs on IFTTT. The study revealed that many users are creating a diverse range of TAP rules, indicating the increasing popularity of TAP in the real world. Early iterations of TAP systems focused on simple, singular relationships between a trigger and an action. However, TAP systems have evolved to support more complex conditions, integrating multiple triggers and actions (Ur et al. [2014]), and considering context and user preferences (Ghiani et al. [2017]).

## 2.2 Conflicts in TAP

TAP rules are prone to bugs, and users find distinguishing between events and states difficult, leading to confusion.

As mentioned in the previous chapter, TAP rules are prone to bugs, leading to conflicts in smart home use. A paper by Huang and Cakmak [2015] demonstrated that users find distinguishing between events and states difficult. Events are specific occurrences at a particular moment, while states are conditions that remain true over an interval of time. This distinction is crucial because it can lead to confusion when defining TAP rules (Brackenbury et al. [2019]). Brackenbury et al. [2019] identified and categorized ten classes of

[3]https://www.samsung.com/us/smartthings (last accessed on February 2, 2024)

[4]https://www.philips-hue.com (last accessed on February 2, 2024)

TAP bugs, grouping them under control flow bugs, timing bugs, and inaccurate user expectations.

| **Unintended Behavior** | **Description** |
|---|---|
| Action duration disregarded | Rule does not take into account action's duration |
| Action incorrect | Action is configured to operate outside entity's capabilities |
| Action reversal | Action is automatically reversed by another rule |
| Action reversal missing | Action is never reversed by another rule |
| Actions inconsistent | Contradictory actions on same entity |
| Actions redundant | Similar actions executed on same entity |
| Condition bypass | Other rule with less strict condition is executed instead |
| Condition unsatisfiable | Condition can never evaluate to true |
| Effects inconsistent | Contradictory effects on same environmental property |
| Entity unavailable | Action cannot be executed when entity is no longer accessible |
| Events redundant | Multiple events created by same sensor in a specific time window |
| Loop | Actions of multiple rules cyclically trigger each other |
| Race condition | Execution order affects which rules are triggered |
| Rule incomplete | Rule does not consider all possible sensor values |
| Rule set overtrust | User trusts set of rules that is not configured properly |
| Trigger incomplete | Some trigger parameters are not specified |
| Triggers redundant | Trigger is activated multiple times by similar events |
| Trigger overtrust | User trusts trigger that is not configured properly |
| Violated constraint | Violation of user-defined boolean expression |

**Table 2.1:** Coppers et al. [2022] created a comprehensive list of 19 different categories of unintended actions by compiling various taxonomies. (Table taken from Coppers et al. [2022])

As detailed by Brackenbury et al. [2019], control flow bugs include three types. First, the infinite loop bug occurs when rules trigger each other continuously, similar to infinite loops in traditional programming languages. Both static and dynamic analyses can detect this bug. Second, contradictory action bugs represent scenarios where rules conflict over time, such as a system alternating between heating and air conditioning without stabilizing. Third, repeated

Brackenbury et al. [2019] identify three control flow bug types: infinite loop, contradictory action, and repeated triggering.

triggering happens when a rule is expected to activate once but triggers multiple times. Static analysis can identify potential cases of repeated triggering but may not always determine if the triggering is intentional.

Timing bugs have two types, nondeterministic timing, and extended action bugs, and can be identified through static and dynamic analyses.

Timing bugs comprise two types (Brackenbury et al. [2019]). Nondeterministic timing bugs arise from the unpredictable order in which a system processes simultaneous triggers. Both static and dynamic analyses can identify these bugs. Extended action bugs occur with actions extending over time rather than instantaneous. For instance, a coffee brewing rule might cause excessive coffee production if it reactivates during the brewing process.

Inaccurate user expectation bugs comprise missing reversals, secure-default bias, time-window fallacy, priority conflicts, and flipped triggers.

Inaccurate user expectations comprise five bug types (Brackenbury et al. [2019]). One is the missing reversal, where users fail to specify a rule for reversing an action, expecting automatic reversion, a common misconception. Static analysis can detect but not always correct these bugs. Secure-default bias bugs occur when users incorrectly assume systems default to a safe state. The time-window fallacy involves users misinterpreting the time window in a ruleset, particularly in complex temporal rules. Priority conflict bugs emerge when users struggle to prioritize multiple rules affecting the same device. Finally, flipped triggers occur when users mix up a trigger's event and state components, resulting in rules that behave unexpectedly.

Coppers et al. [2022] compiled a list of 59 conflict types and categorized them into 19 unique unintended behaviors.

The bug types identified by Brackenbury et al. [2019] present the common TAP bugs. Meanwhile, Coppers et al. [2022] compiled a list of 59 conflict types based on the works of Al Farooq et al. [2019], Brackenbury et al. [2019], Ma et al. [2017], Magill and Blum [2016], and Wang et al. [2019]. They then categorized these conflicts into 19 unique unintended behaviors, as detailed in Table 2.1. Coppers et al. [2022] noted that the table they presented needs to be more comprehensive. They expect to discover additional conflict types as the trigger-condition-action paradigm evolves and becomes more advanced.

## 2.3 Solutions using Static Analysis

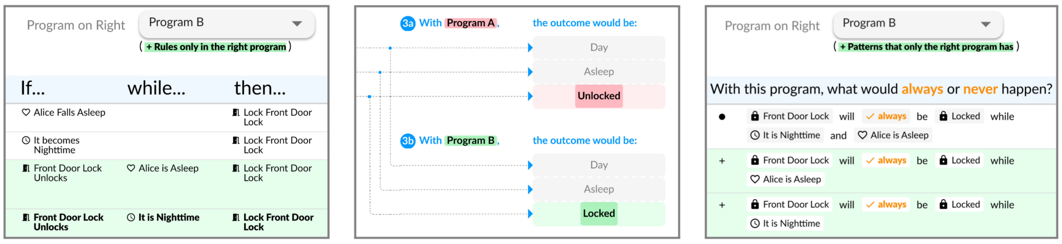
A study by De Russis and Monge Roffarello [2018] presents a debugging approach using Semantic Web and Petri Nets to assist users in defining trigger-action rules for smart devices and web services. The approach involves translating rules into corresponding Petri nets to detect loops, inconsistencies, and redundancies in rules, followed by analysis for conflict detection. The study's evaluation suggests that users successfully used this debugging method to identify and understand inconsistencies and loops in TAP rules. However, users perceived redundancies as less critical, sometimes choosing to retain them.

De Russis and Monge Roffarello [2018] proposed a debugging method using Semantic Web and Petri Nets for trigger-action rules, successfully identifying inconsistencies and loops.

Building on the works of De Russis and Monge Roffarello [2018], Corno et al. [2019] designed *EUDebug* to assist end users in debugging trigger-action rules by identifying conflicts and simulating rule behavior. *EUDebug* automatically detects potential issues during rule composition and shows conflicts with existing rules at the end of the process. Its main component is a semantic-colored Petri net (SCPN) for modeling and simulation and a web interface for rule composition and problem analysis. The interface includes features for rule composition, problem checking, and step-by-step explanation, enhancing user understanding of rule conflicts and run-time behavior. It simplifies rule creation and helps users grasp the implications of their configurations. Corno et al. [2019] evaluated *EUDebug* for its effectiveness in helping users understand and identify problems in their trigger-action rules. Participants perceived loops and inconsistencies as significant issues, while redundancies were comparatively more acceptable. The loop was the most challenging problem, leading participants to utilize the step-by-step explanation feature frequently. While only highlighting detected issues was often insufficient, the step-by-step simulation of rules notably aided in understanding the problems.

*EUDebug* helps end-users debug TAP rules by detecting potential issues during rule composition and simulating rule behavior with a web interface.

The paper by Zhao et al. [2020] introduces a set of prototype interfaces designed to visualize and clarify the differences in variants of TAP rules. The interfaces assist users, regardless of their experience, in understanding the dif-



Program on Right Program B

( + Rules only in the right program )

| If... | while... | then... |
|---|---|---|
| Alice Falls Asleep | | Lock Front Door Lock |
| It becomes Nighttime | | Lock Front Door Lock |
| Front Door Lock Unlocks | Alice is Asleep | Lock Front Door Lock |
| **Front Door Lock Unlocks** | **It is Nighttime** | **Lock Front Door Lock** |

3a With Program A, the outcome would be:
Day
Asleep
Unlocked
3b With Program B, the outcome would be:
Day
Asleep
Locked

Program on Right Program B

( + Patterns that only the right program has )

With this program, what would always or never happen?

- Front Door Lock will always be Locked while It is Nighttime and Alice is Asleep
- + Front Door Lock will always be Locked while Alice is Asleep
- + Front Door Lock will always be Locked while It is Nighttime

**Figure 2.1:** TAP *diff* interfaces demonstrate both a conventional method of presenting differences in program text (left) as well as innovative techniques of displaying differences in program outcomes (center) and properties (right). (Image taken from Zhao et al. [2021])

ferences in rule changes. These interfaces offer three levels of granularity: textual differences in the rules, differences in system actions under specific scenarios, and property differences. Inspired by code *diff* utilities and collaborative word-processing tools, these interfaces provide a comprehensive view of TAP rule variants, helping debug and understand the behavior of smart devices and systems. The text difference interface, inspired by GitHub's *diff* views, highlights syntax differences in TAP rules, categorizing rules as "added," "removed," or "modified." The flowchart-based interface assists users in understanding behavioral differences between rules under specific scenarios, like different actions triggered by the same event. It depicts how various system states evolve, highlighting differences in device states and actions taken by each rule. This approach helps in clarifying complex interactions and redundancies in TAP rules. Additionally, the paper introduces a form-based interface for comparing multiple TAP rules. This interface uses a multiple-choice form to present scenario-specific questions, helping users choose between actions taken by different rules. It identifies the rule(s) most aligned with user preferences based on their responses. The interface also explores property differences, comparing long-term system patterns that address the challenge of manually analyzing rules.

Zhao et al. [2020] introduce prototype interfaces that visualize and clarify differences in variants of TAP rules, assisting users in understanding rule changes and system behavior.

In another paper, a user study conducted by Zhao et al. [2021] extended their previous work (Zhao et al. [2020]) on prototype interfaces to visualize the differences in variants of TAP rules by evaluating the interfaces as shown in

Figure 2.1. The user study revealed that users could find differences by examining rules alone for short, simple programs. However, participants using semantic-difference interfaces performed better for longer, more complex programs than those using traditional rules or text-difference interfaces. The outcome-difference flowchart interface was better in tasks requiring the identification of specific differences, while the property-difference interface was better for abstract differences. The study highlights the importance of interfaces to visualize information with various levels of detail.

The study suggests semantic-difference interfaces perform better than traditional rules for more extended and complex programs.

A paper by Yusuf et al. [2022] introduces *RecipeGen*, an approach to generating TAP rules using natural language descriptions, employing a deep learning framework. It diverges from previous studies by framing TAP rules as a sequence learning task rather than a classification one, utilizing Transformer sequence-to-sequence architecture. This method allows *RecipeGen* to capture the intricate relations among various elements of the rules, leading to more accurate and relevant rule generation. The key to *RecipeGen's* effectiveness is its use of pre-trained autoencoding models, which enhances its understanding of rule descriptions. *RecipeGen* evaluation demonstrates high performance in generating TAP rules. This advancement marks a significant contribution to the field, offering to avoid conflicts during rule creation.

*RecipeGen* uses deep learning to generate accurate TAP rules from natural language descriptions, resulting in more relevant and conflict-free rule creation.

*AutoTap* by Zhang et al. [2019] helps create TAP rules by allowing users to define properties and rules through a graphical interface, making it more accessible to non-technical users. It automates the creation and repair of TAP rules to ensure compliance with specified properties provided by the user. This feature reduces the risk of human error in TAP rules. *AutoTap* uses linear temporal logic to translate user-specified properties into formal models. It then synthesizes TAP rules that are compliant, accommodating, and valid using an algorithm. This approach ensures that the smart home system adheres to user-defined properties while reducing TAP rule conflicts.

*AutoTap* automates creating and repairing TAP rules, reducing human error and ensuring compliance with user-defined properties.

*Trace2TAP* by Zhang et al. [2020] also helps in TAP rule creation. However, it analyzes user behavior traces to gener-

*Trace2TAP* generates TAP rules by analyzing user behavior traces, helping users align device automation with their preferences.

ate TAP rules. It identifies device actions to be automated that align with the user's intent and uses symbolic execution and SAT-solving to create relevant TAP rules. The system clusters and ranks these rules to help users select the most suitable ones. *Trace2TAP* was evaluated through a field study in office settings, showing its effectiveness in aligning user intent with automated actions. Additionally, *Trace2TAP* helps in debugging by generating patches to refine existing rules based on user interactions, further aligning device automation with user preferences.

## 2.4 Solutions using Dynamic Analysis

*FORTNIoT* predicts smart home behavior by simulating TAP rules and self-sustaining predictions, helping users better understand and manage their smart home systems.

Coppers et al. [2020] introduce *FORTNIoT*, a tool designed to predict the future behavior of smart homes using existing TAP rules. It simulates these rules in conjunction with self-sustaining predictions, like weather forecasts or sun positions, to deduce the future states of the smart home. *FORTNIoT's* algorithm integrates these predictions with smart home rules, enabling users to understand the complex interactions within their smart home systems. The process involves continuously updating predictions based on real-time data and simulating various scenarios. A between-group study by Coppers et al. [2020] revealed that *FORTNIoT* helps non-experienced users better understand and feel more confident about smart home behaviors. It shifts user responsibility from assembling answers to simply verifying them. They remarked that predictions improve accountability and comprehension of smart homes, help resolve conflicts, and manage smart home behaviors. Additionally, they highlighted that visualization tools help troubleshoot past malfunctions and could benefit future smart home behavior research.

*FortClash* uses a two-step process and breadth-first algorithm to predict unintended smart home behaviors.

Expanding on their work on *FORTNIoT*, Coppers et al. [2022] introduces *FortClash*, which uses a two-step process for predicting unintended smart home behaviors. Initially, it robustly forecasts the smart home's future behavior using TAP and external prediction services, like weather forecasts. Then, unlike *FORTNIoT's* depth-first approach, it employs a breadth-first algorithm to detect potential unin-

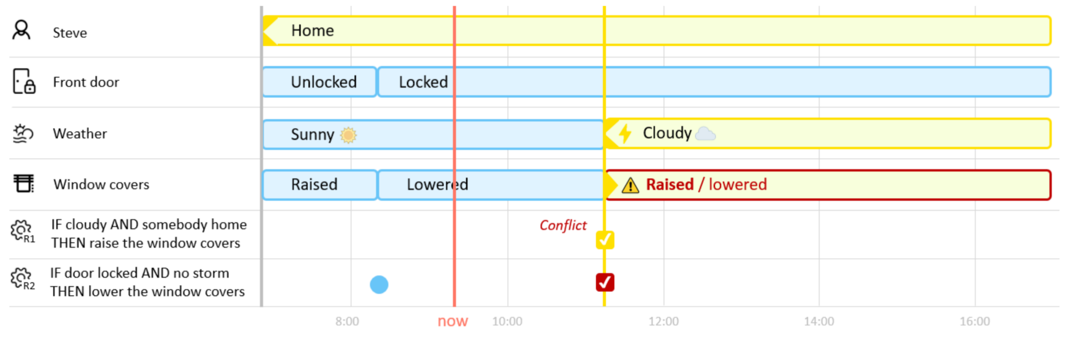


**Figure 2.2:** *FortClash* predicts and allows smart home users to temporarily suspend one-time exceptions for a rule's action at a specific moment. (Image taken from Coppers et al. [2022])

tended behaviors accurately. This algorithm ensures that all potential outcomes of the rules are considered independently and sequentially, thus improving the accuracy of the predictions. Users can then identify potential issues through a dashboard, as shown in Figure 2.2. *FortClash* examines the predicted states, rule executions, and a causality graph to spot unintended behaviors like loops or inconsistencies. If a state participates in multiple unintended behaviors, *FortClash* merges them until resolution. Users can control outcomes through a visualization interface, which allows them to suppress specific actions. For example, unchecking a box linked to an action prevents its execution, removing its effects from future predictions. This process feeds back into the prediction algorithm, which maintains a list of suppressed actions to ensure they do not reoccur in future simulations. *FortClash* also allows users to modify action parameters instead of simply suppressing them. Users can adjust parameters like partially raising window covers and delaying action execution. These temporary modifications are stored in a list and considered by the prediction algorithm, overriding the default action parameters. This change helps meet the current needs without altering the rule's overall settings for future executions. *FortClash* also allows users to manually plan action executions, addressing situations where rules fail to trigger. Users can select execution times directly on the timeline visualization, using configurations of existing actions or setting new ones.

*FortClash* lets users control, modify, and plan actions through a visualization interface to address predicted unintended behaviors.

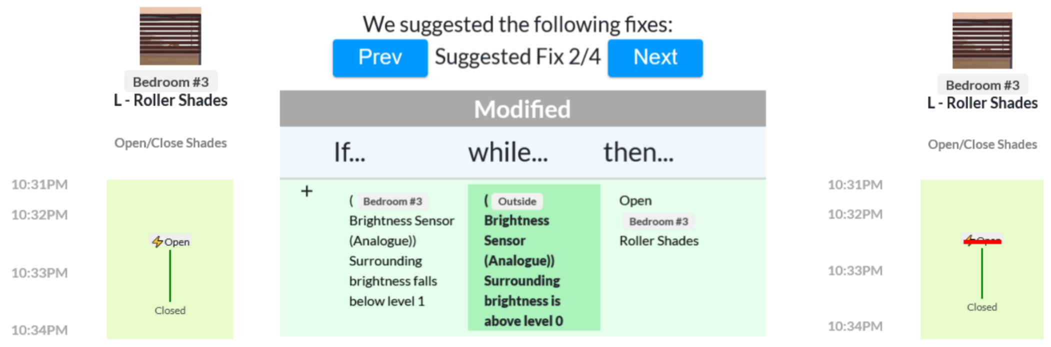

Bedroom #3
L - Roller Shades
Open/Close Shades
10:31PM
10:32PM
10:33PM
10:34PM
Open
Closed
We suggested the following fixes:
Prev
Suggested Fix 2/4
Next
Modified
If...
while...
then...
+
( Bedroom #3 Brightness Sensor (Analogue)) Surrounding brightness falls below level 1
( Outside Brightness Sensor (Analogue)) Surrounding brightness is above level 0
Open Bedroom #3 Roller Shades
Bedroom #3
L - Roller Shades
Open/Close Shades
10:31PM
10:32PM
10:33PM
10:34PM
Open
Closed

**Figure 2.3:** Debugging workflows by Zhang et al. [2023] provide a history visualization interface (left), patch synthesis output interface (center), and patch behavior visualization (right). (Image taken from Zhang et al. [2023]

Users can also ignore system warnings about potential unintended behaviors once or always. Ignored instances are recorded, and future matching predictions are updated accordingly.

*Explicit-Feedback* and *Implicit-Feedback* workflows by Zhang et al. [2023] outperformed the *Control* workflow in debugging TAP rules.

The paper by Zhang et al. [2023] introduces new interfaces and algorithms for debugging TAP rules, focusing on the end-to-end process of resolving conflicts in smart homes. These interfaces include identifying unexpected behaviors, locating faults in TAP rules, proposing modifications, and refining these changes. The authors developed two workflows, *Explicit-Feedback* and *Implicit-Feedback*, each using different methods for gathering user feedback on system misbehavior. A *Control* workflow was also implemented, representing traditional manual TAP debugging without tool support. The *Explicit-Feedback* workflow uses user annotations on automation misbehaviors to suggest corrective patches for the rules. Users mark incorrect automation via a history visualization interface, and the system generates potential patches, as shown in Figure 2.3. These patches are then presented to the user with detailed visualizations for informed decision-making in selecting and refining patches. On the other hand, the *Implicit-Feedback* workflow automatically infers system misbehaviors by analyzing user interactions, such as manual device actuation or reversing automation. Unlike the *Explicit-Feedback* workflow, it does not require users to annotate misbehav-

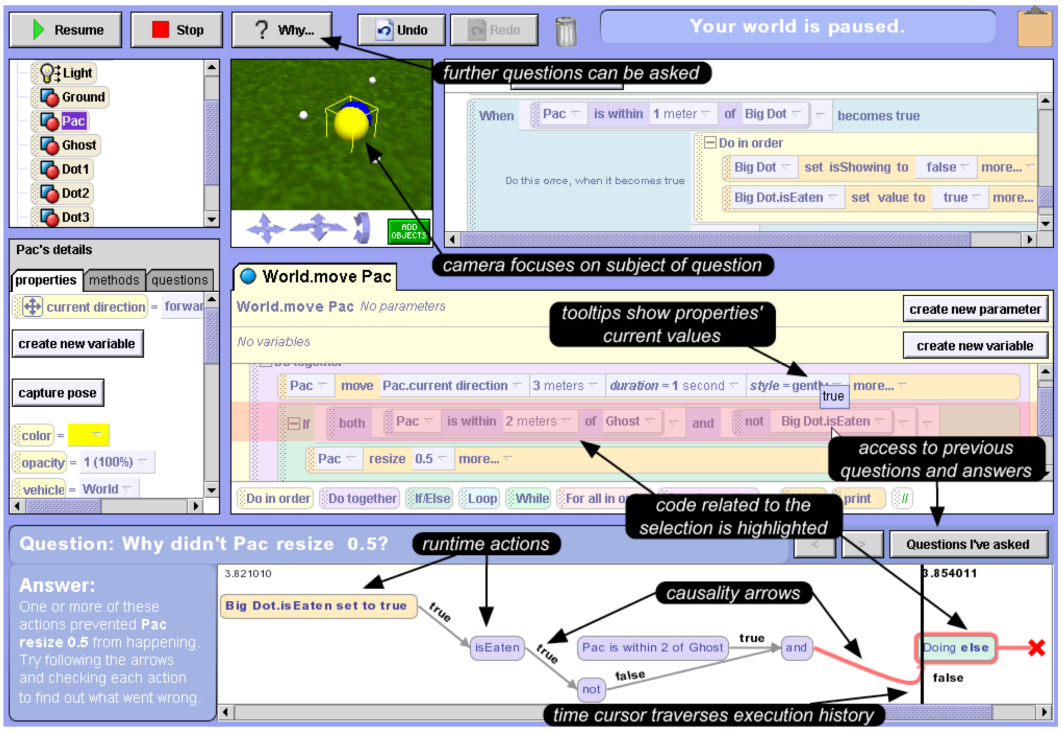

Resume
Stop
Why...
Undo
Redo
Your world is paused.
further questions can be asked
camera focuses on subject of question
tooltips show properties' current values
access to previous questions and answers
code related to the selection is highlighted
runtime actions
causality arrows
time cursor traverses execution history
Question: Why didn't Pac resize 0.5?
Questions I've asked
Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.
Big Dot.isEaten set to true
isEaten
Pac is within 2 of Ghost
not
and
Doing else
true
false

**Figure 2.4:** The answer provided by *Whyline* displays a visual representation of the runtime actions at the bottom of the figure. Users can utilize the time cursor to navigate through the execution history. (Image taken from Ko and Myers [2004])

iors explicitly. This workflow identifies potential issues like under-automation (automated events that should have occurred but did not) and over-automation (unnecessary automated events). Users still get to decide which device actions are incorrect, helping accurately identify false negative and false positive misbehaviors. Upon evaluation, Zhang et al. [2023] found that the *Explicit-Feedback* and *Implicit-Feedback* groups performed better than the *Control* group in problem-solving tasks. Without tool support, the *Control* group struggled more, particularly in complex tasks. However, usability scores were similar across all interfaces, but the *Implicit-Feedback* group required fewer clarifications. Time spent on tasks varied, with the *Explicit-Feedback* interface requiring more time due to additional feedback steps.

Zhang et al. [2023] pointed out two categories of TAP conflicts: over-automation (unnecessary automated events)

*Whyline* helps programmers debug code by allowing them to ask specific questions about runtime bugs, which significantly reduces debugging time.

and under-automation (automated events that should have occurred but did not). The first question that comes to users' minds is why over-automation actions occur and why under-automation events do not occur. Ko and Myers [2004] answered similar questions with their prototype *Whyline*. *Whyline* introduces an interrogative debugging paradigm to debug code, allowing programmers to ask specific *why did* or *why didn't* questions about runtime bugs. *Whyline* visualizes runtime events for the slice of the program relevant to these questions, along with connected arrows to represent data and control flow causality, as shown in Figure 2.4. A user study by Ko and Myers [2004] showed that *Whyline* significantly reduced debugging time for programmers.

# Chapter 3

# The Debugging Dashboard

This chapter will discuss the *Debugging Dashboard* designed as part of this thesis. We will start by providing an overview of the application that incorporates the dashboard in Section 3.1. Then, we will discuss the *Debugging Dashboard* interface and its design choices in Section 3.2. Finally, we will discuss implementing the application and the dashboard in detail.

## 3.1 Overview

Our application simulates conflict situations in TAP rules, offering the *Debugging Dashboard* to identify the causes of those conflicts.

We implemented a web application to create virtual smart home setups, simulating conflict situations in TAP rules and offering the *Debugging Dashboard* to identify the causes of these conflicts. The application allows for creating and simultaneously running various smart home configurations with different conflict scenarios. The application is implemented in a client-server architecture, as shown in Figure 3.1. The client-side developed using React[1], a Javascript[2] library, offers an interactive user interface for managing and debugging smart homes. It primarily in-

[1]https://react.dev (last accessed on February 2, 2024)
[2]https://www.javascript.com (last accessed on February 2, 2024)

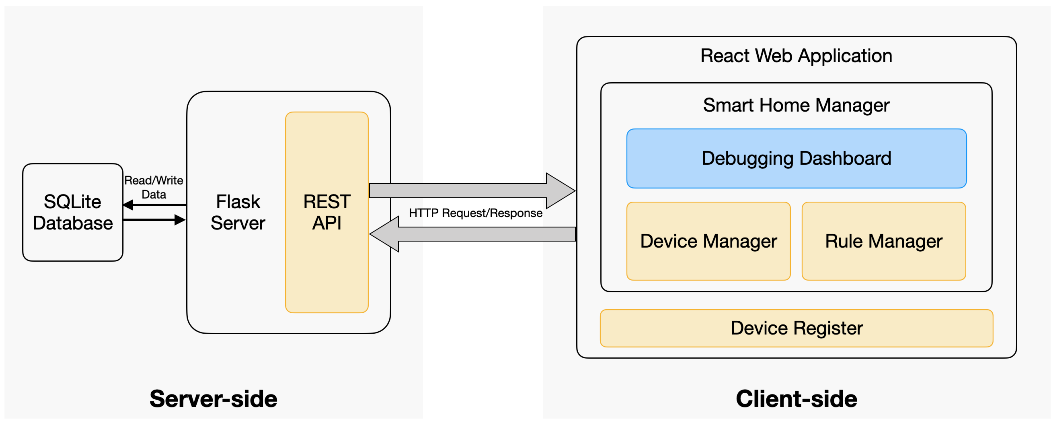


**Figure 3.1:** System architecture of our web application that offers the *Debugging Dashboard*. It shows the interaction between client-side components, server-side components, and the database and data flow within the smart home debugging system.

cludes the *Smart Home Manager* and the *Device Register*. The *Device Register* allows us to add virtual smart devices to the application. The added devices can then be used when configuring the virtual smart homes. The *Smart Home Manager*, on the other hand, includes the following components:

**a.** *Debugging Dashboard*: It displays the activated rules in the smart home to assist in finding the cause of the rule conflicts. We will discuss design choices in Section 3.2 and implementation of the dashboard in Section 3.3.1.

**b.** *Device Manager*: It manages devices within the smart home ecosystem, allowing users to add and modify device settings.

**c.** *Rule Manager*: It helps create TAP rules, providing an interface for users to define triggers, conditions, and actions for their smart home devices.

The client communicates with the server side via Hypertext Transfer Protocol (HTTP) requests to a Representational State Transfer (REST) Application Programming Interface (API). The server-side is implemented in Flask[3],

[3]https://flask.palletsprojects.com (last accessed on February 2, 2024)

a lightweight and flexible Python[4] web framework. The Flask API processes client requests, handling the application's logic and interacting with the SQLite[5] database. This database stores data related to smart home devices, TAP rules, and other relevant information needed for debugging. The architecture is designed to manage and debug smart home rules efficiently.

Before moving on to the interface design, we must consider the context of using the application and the debugging dashboard. We should determine which TAP rules and devices to use in the application. This is important to ensure the dashboard can handle the complexities and variations in smart home systems.

As discussed in Section 2.2, distinguishing between events and states can be confusing in TAP rules. Therefore, for clarity, we use TAP rules with a single event-based trigger and a single event-based action, and an optional condition group, when present, groups multiple state-based conditions in the *AND*/*OR* chain.

We use TAP rules with a single trigger, single action, and optional condition groups for multiple conditions.

A smart device can have sensors, actuators, or both (Hribernik et al. [2011]). Sensors collect environmental data, such as temperature or light brightness levels, while actuators perform actions based on this data, like adjusting thermostats or closing blinds. Our application supports smart devices with sensors, actuators, or a combination of both. We use the common term *Device_property* to refer to both sensors and actuators.

Our application supports smart devices with sensors, actuators, or both.

## 3.2 Debugging Dashboard Design

The *Debugging Dashboard* serves as the core component of our application. This section looks into the interface's elements, the reasoning behind their design, and their role in supporting the debugging process.

[4]https://www.python.org (last accessed on February 2, 2024)

[5]https://www.sqlite.org/index.html (last accessed on February 2, 2024)

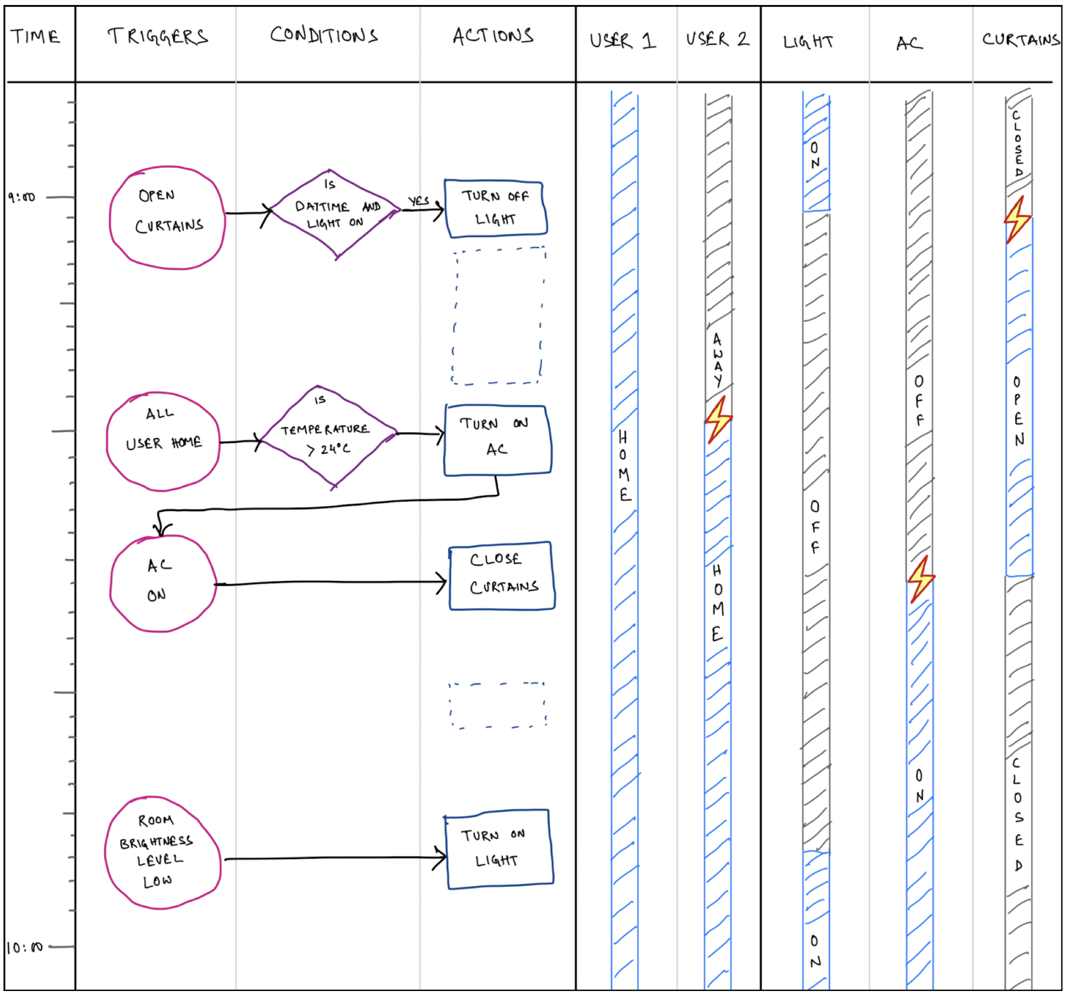


**Figure 3.2:** Initial Prototype of the *Debugging Dashboard*. This early version of the dashboard shows a timeline of TAP rule events, visual representations for triggers, conditions, and actions, and a display of smart home devices with status changes over time.

### 3.2.1 Initial Prototype

We created a dashboard prototype, including a timeline of TAP rule events and device status change visualization.

We aimed to create an effective dashboard interface that implements trace-based debugging, as defined in Chapter 1, to assist users in debugging TAP rules in their smart home environments. Our focus was on integrating essential elements that would make the process easier. To meet our requirements, we designed an initial prototype, as shown in Figure 3.2, that includes a timeline displaying TAP rule events chronologically for users to follow the sequence of triggers, conditions, and actions. We represented each TAP rule event, including triggers, conditions, and actions, using different shapes to make them distinguishable. We also added a visualization of all smart home devices and their status changes, which is crucial in showing how each device's state changes and correlates with activating TAP rules. Although we included a visualization for all user statuses in our prototype, we excluded it in our final dashboard to keep the interface simple and easy to debug.

### 3.2.2 Final Interface

The final interface includes some minor changes to improve the debugging process.

When designing our final dashboard interface, we made some minor adjustments to the initial prototype while still retaining most of the original design elements. As mentioned earlier, we decided to exclude the visualization for user statuses. We refined the representation of TAP rule events by using rectangular boxes instead of different shapes, which makes it easier to read the textual information inside those shapes while still being distinguishable based on the column headers provided for each column. Additionally, we added labels to make it visually clear when the device value changes. To provide additional details, we included mouse hover options. In the following paragraphs, we will explain the detailed design of each component.

The dashboard interface is divided into two main sections: the *Rule Timeline* and the *Device Timeline*.

The final dashboard interface is divided into two main sections: the *Rule Timeline* (the left half) and the *Device Timeline* (the right half), as shown in Figure 3.3. The *Rule Timeline* displays the activated rules in the order of their execution, following the visual approach of *Whyline's* methodology (Ko and Myers [2004]). On the other hand, the *Device Timeline* is inspired by *FortClash's* (Coppers et al. [2022]) and Zhang et al. [2023]'s techniques, which illustrate the status of connected smart devices over time. This layout ensures that the execution of rules and the reaction of devices to these rules are clearly understood, helping in the debugging process.

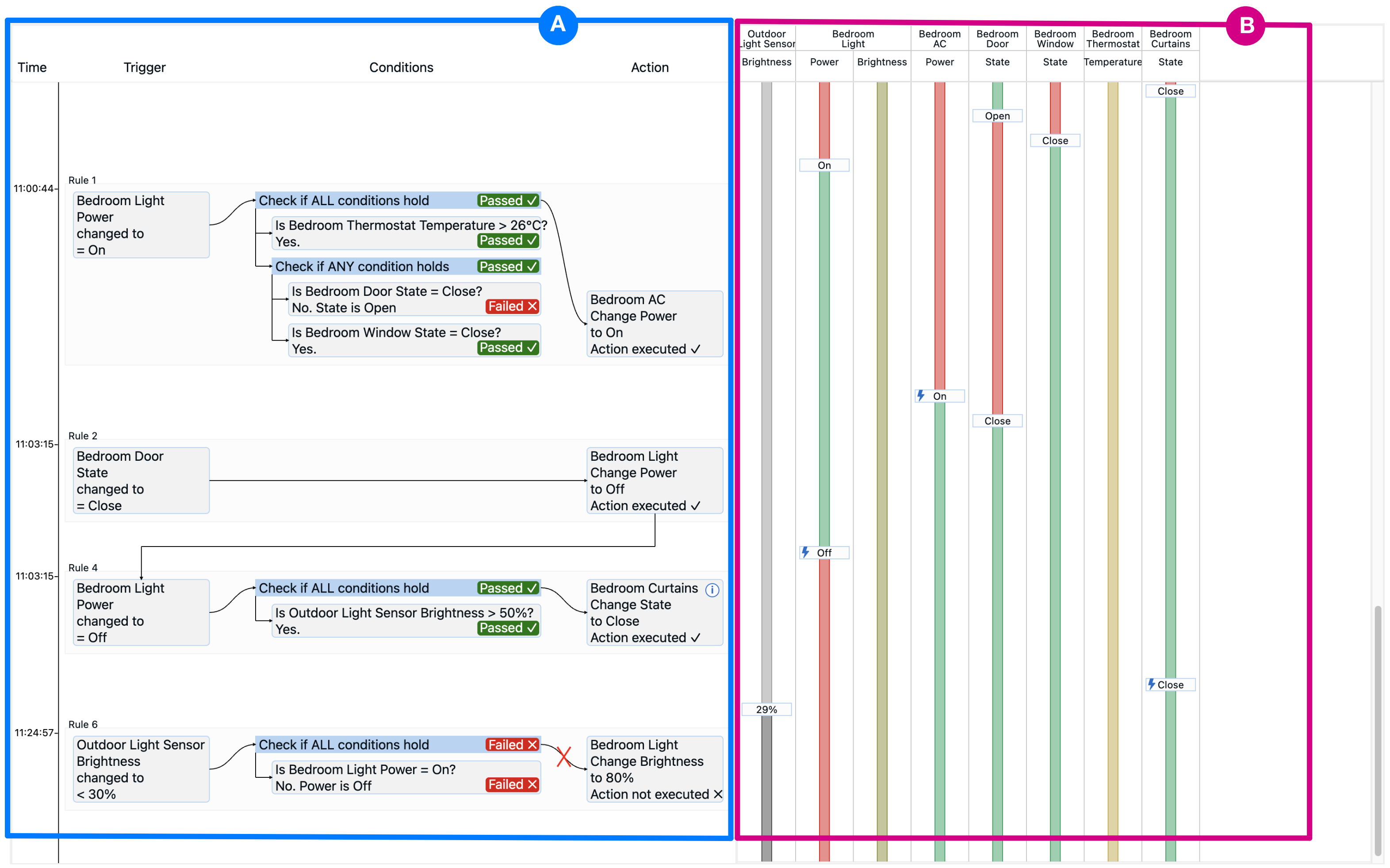
The *Rule Timeline* displays the sequence of rule activations over time, with a visual grouping of rule components and arrows indicating the control flow between different rules.

The dashboard's leftmost column displays the timestamps for each activated rule, arranged horizontally on the *Rule Timeline*. This design enables users to observe the sequence of rule activations over time easily. The timestamps are not displayed to scale to minimize empty spaces between rules that are activated at widely different times and to provide additional space to avoid overlapping of simultaneously activated rules. The *Rule Timeline* portrays each component of the rule—trigger, action, conditions, and condition groups—as separate events within a unified box. This visual grouping indicates that they jointly form a single rule. Arrows link these components to display the sequence and causality of rule execution. Furthermore, actions that activate other rules are connected to the triggers of subsequent rules with arrows, demonstrating the control flow between different rules.

Each rule on the dashboard has a display box with the rule name shown at the top-left corner, clarifying its function. The information displayed for all rule components reflects the information needed to form the rule, ensuring that users understand the rules.

The trigger box shows data in the format of "*<Room_name> <Device_name> <Device_property>* changed to *<operator> <new_state>*," for example, "*Bedroom Light Brightness changed to <= 70%.*" The action box shows data in the format of "*<Room_name> <Device_name>* change *<Device_property>* to *<new_state>*," for example, "*Bedroom Light Change Brightness to 70%.*" The action box also displays "*Action executed* ✓" if the action was carried out successfully and "*Action not executed* ×" if the action failed due to failed conditions.

The condition box shows data in the format of "Is *<Room_name> <Device_name> <Device_property> <operator> <condition_state>*?," for example, "*Is Bedroom Light Brightness <70%?*" Additionally, the condition box informs the user if the condition is met. It displays "*Yes.*" if the condition is met, and "*No. <Device_property>* is *<Current_state>*" otherwise, for example, "No. Brightness is 80%."



11:00:44 Rule 1

Bedroom Light Power changed to = On

Check if ALL conditions hold Passed ✓

Is Bedroom Thermostat Temperature > 26°C? Yes. Passed ✓

Check if ANY condition holds Passed ✓

Is Bedroom Door State = Close? No. State is Open Failed ✕

Is Bedroom Window State = Close? Yes. Passed ✓

Bedroom AC Change Power to On Action executed ✓

11:03:15 Rule 2

Bedroom Door State changed to = Close

Bedroom Light Change Power to Off Action executed ✓

11:03:15 Rule 4

Bedroom Light Power changed to = Off

Check if ALL conditions hold Passed ✓

Is Outdoor Light Sensor Brightness > 50%? Yes. Passed ✓

Bedroom Curtains Change State to Close Action executed ✓

11:24:57 Rule 6

Outdoor Light Sensor Brightness changed to < 30%

Check if ALL conditions hold Failed ✕

Is Bedroom Light Power = On? No. Power is Off Failed ✕

Bedroom Light Change Brightness to 80% Action not executed ✕

**Figure 3.3:** The user interface of the *Debugging Dashboard* highlights the *Rule Timeline* (A), which displays the activated rules in the order of their execution, and the *Device Timeline* (B), which shows the status of connected smart devices over time.

*Condition Status Indicator* is designed to help users quickly determine if conditions are met and whether an action was executed.

Boxes representing *AND* condition groups display *"Check if ALL conditions hold."* Conversely, boxes for *OR* condition groups show *"Check if ANY condition holds."* A *"Passed"/"Failed"* indicator highlighted with green and red backgrounds, known as the *Condition Status Indicator*, for each condition and condition groups, provides a quick visual cue to users, enabling them to quickly determine whether specific conditions were met without reading through all the text. Additionally, the interface features a red cross over the arrow that connects a failed condition group to its corresponding action. This indicates that the action was not executed due to the failure of the condition group, helping the user understand the rule execution process.

The *Device Timeline* displays the state of each *Device_property* in separate columns, which change over time.

The *Device Timeline* displays the state of each *Device_property* in separate columns, which change over time. At the top of each column, the *Device_name* and the corresponding *Device_property* are displayed. A small box called the *Device State Label* showing the new value is marked whenever a *Device_property* changes state. *Device State Labels* for automated state changes also include a flash icon (⚡) to differentiate them from manual or environmentally triggered state changes. A thick vertical line colored uniquely according to the state value extends from the *Device State Label* to represent the unchanged state over time until the following state change occurs. Unique colors are assigned to each distinct state of *Device_properties* with fixed states, such as Power (On/Off), while for *Device_properties* with a range of states, e.g., Brightness ranging from 0% to 100%, extreme values are assigned unique colors, which are interpolated for intermediate values within the range. The design is inspired by Zhang et al. [2023].

The dashboard's timeline is not to scale, which helps to avoid overlapping visual elements.

As mentioned earlier, the timeline is not to scale. This design also prevents overlapping *Device State Labels* in cases where changes occur in rapid succession. Furthermore, the *Device State Label* that triggers a rule is displayed slightly above the corresponding rule on the *Rule Timeline* of the dashboard. This visual difference can help users infer the sequence of events, as the timestamps are nearly identical. Similarly, state changes in *Device_properties* resulting from a rule's action are visually positioned slightly lower, helping

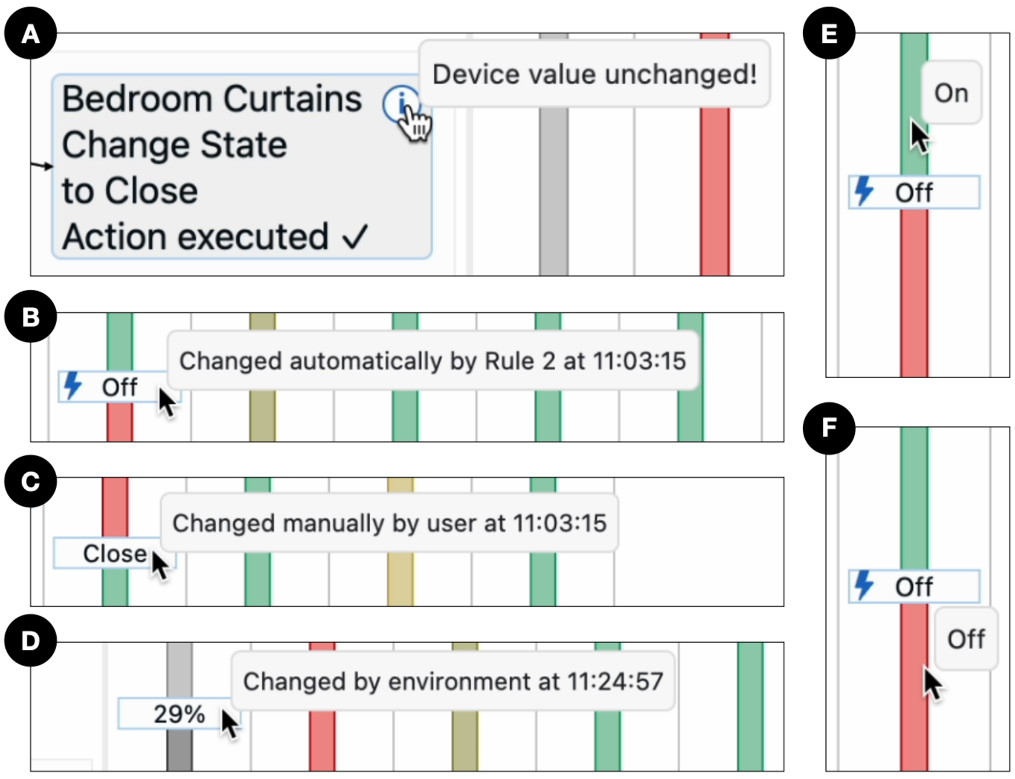
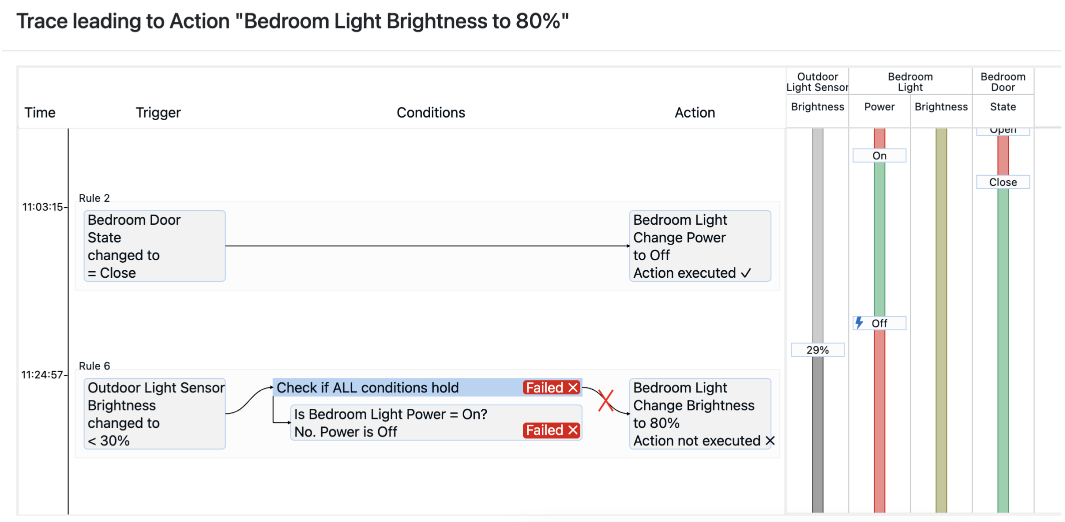

A
Bedroom Curtains
Change State
to Close
Action executed ✓
Device value unchanged!
B
Off
Changed automatically by Rule 2 at 11:03:15
C
Close
Changed manually by user at 11:03:15
D
29%
Changed by environment at 11:24:57
E
On
Off
F
Off
Off

**Figure 3.4:** Additional information is displayed to assist with debugging when hovering over different components. (A) An information icon appears in the action box's top right corner if a rule's action does not modify the state of the *Device_property*. (B, C, and D) Hovering over the *Device State Labels* will reveal the cause and exact timestamp of the change. (E and F) Display the state of the *Device_property* before and after a *Device State Label*.

understand the cause-and-effect relationship.

We display additional information using the mouse hover feature to enhance the debugging process, as depicted in Figure 3.4. Hovering over *Device State Labels* reveals the change's cause and exact timestamp. For automated changes in *Device_properties*, the display reads *"Changed automatically by <Rule_name> at <Timestamp>,"* like *"Changed automatically by Rule 2 at 12:34:56."* Manual changes show *"Changed manually by user at <Timestamp>,"* while environmental changes display *"Changed by environment at <Timestamp>."* Additionally, when a rule's action does not alter the *Device_property* state (e.g., a light turned on by a rule

The mouse hover feature provides additional information for debugging.



Rule 2

Bedroom Door State changed to = Close

Bedroom Light Change Power to Off Action executed ✓

Rule 6

Outdoor Light Sensor Brightness changed to < 30%

Check if ALL conditions hold Failed ✕

Is Bedroom Light Power = On? No. Power is Off Failed ✕

Bedroom Light Change Brightness to 80% Action not executed ✕

**Figure 3.5:** Dashboard's *Filtered View* displays only the relevant rules and devices involved in the success or failure of a particular action.

when it was already on), an information icon appears in the action box's top right corner, displaying "*Device value unchanged!*" upon hovering.

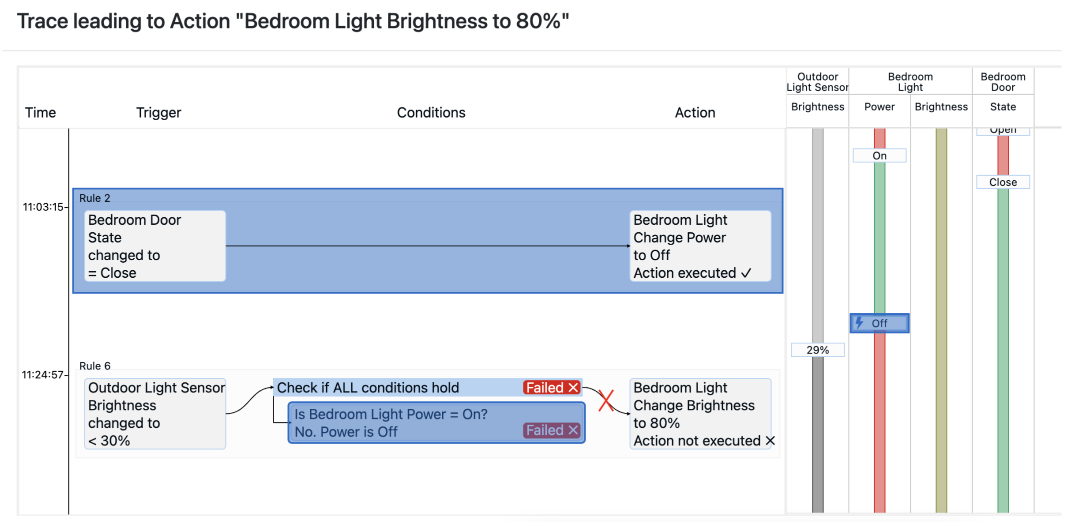
### 3.2.3 Data Filtering

The dashboard has a data filtering feature to help users identify the cause of over and under-automation events.

The dashboard displays a detailed view of the smart home's TAP rules and smart device events. To simplify debugging, we introduced a data filtering feature inspired by *Whyline* (Ko and Myers [2004]). This feature allows users to focus on specific actions, presenting a *Filtered View*, illustrated in Figure 3.5, that traces only the rules and devices involved in that action's success or failure. This filtering approach aims to answer why actions of over-automation occur and under-automation events do not, helping users quickly identify the cause of the issue.

The *Filtered View* supports debugging by highlighting trigger, condition, and action changes, helping users trace issues more effectively.

In the *Filtered View*, the dashboard enhances the debugging process by enabling highlighting based on trigger, condition, and action. Selecting a trigger box highlights the corresponding *Device_property* state change that initiated the rule. Similarly, clicking an action box highlights the resul-


Trace leading to Action "Bedroom Light Brightness to 80%"

| Time | Trigger | Conditions | Action |
|---|---|---|---|
| 11:03:15 | Rule 2<br>Bedroom Door State changed to = Close | | Bedroom Light Change Power to Off<br>Action executed ✓ |
| 11:24:57 | Rule 6<br>Outdoor Light Sensor Brightness changed to < 30% | Check if ALL conditions hold Failed ✕<br>Is Bedroom Light Power = On? No. Power is Off Failed ✕ | Bedroom Light Change Brightness to 80%<br>Action not executed ✕ |

**Figure 3.6:** In *Filtered View*, clicking trigger, condition, or action highlights relevant information for debugging. The figure shows that clicking on the failed condition highlights the latest *Device State Label* of the *Device_property* in the failed condition and the responsible rule.

tant *Device_property* state change. For conditions, selecting a box highlights the most recent state change of the *Device_property* used in the condition evaluation. Additionally, if an earlier activated rule caused that state change, then that rule is also highlighted. This feature will help users trace the root cause of issues more effectively.

## 3.3 Client-side Implementation

The client-side of the application is implemented using React[6], as mentioned in Section 3.1 for building user interfaces, and Bootstrap[7], a well-known CSS framework. This combination helped develop custom components needed for our design and ensured a responsive, consistent interface. This section describes implementing two primary components: the *Smart Home Manager* and the *Device Register*. The *Smart Home Manager* allows the configuration

[6]https://react.dev (last accessed on February 2, 2024)
[7]https://getbootstrap.com (last accessed on February 2, 2024)

of multiple virtual smart homes to simulate different TAP conflict scenarios and offers the *Debugging Dashboard*. The *Device Register* adds virtual smart devices into the application, which can be used in these smart home setups. The client is accessible at `http://client-url/` (e.g., `http://localhost:3000/` during the user study).

### 3.3.1 Smart Home Manager

The *Smart Home Manager* consists of the *Debugging Dashboard*, the *Home and Device Manager*, and the *Rule Manager*, as described in Section 3.1.

#### Debugging Dashboard

The *Debugging Dashboard* is implemented using D3 JS library.

The *Debugging Dashboard* for each smart home can be accessed at `http://client-url/case/homeId`, where `homeId` is the unique identifier of the home. The dashboard was developed using D3 JS[8], a JavaScript library known for its flexibility in creating custom data visualizations. By utilizing this library, we were able to develop a custom visualization that met our dashboard interface design requirements, as detailed in Section 3.2. Data for the dashboard is retrieved via an `HTTP GET` request to `http://server-url/api/unfiltered-trace-data/homeId`, `server-url` is described in Section 3.4. The response data, described in Section 3.4.1, is then visualized using D3 JS.

A boolean variable `canFilter` controls the activation of the *Filter* feature, as described in Section 3.2.3, allowing us to evaluate the feature during user study. On clicking the action of the activated rules, the filtered information is visualized in fullscreen Bootstrap `Modal`. The filtered information is retrieved via an `HTTP GET` request to `http://server-url/api/trace-data/homeId/ruleId`, where `ruleId` is the unique identifier of the activated rule. The response data is described in Section 3.4.1.

[8]https://d3js.org (last accessed on February 2, 2024)

When a user clicks on the action of an activated rule, the filtered information is displayed in a full-screen Bootstrap `Modal`. This data is fetched using an `HTTP GET` request to `http://server-url/api/trace-data/homeId/ruleId`, where `ruleId` is the unique identifier of the activated rule. The response data from this endpoint are detailed in Section 3.4.1. This method ensures that users can access detailed, rule-specific information without leaving or reloading the *Debugging Dashboard* page.

### Home and Device Manager

The *Home and Device Manager* supports the creation of virtual smart homes, enabling users to add rooms and devices and update device values to simulate different smart home scenarios. New homes are registered through a form at `http://client-url/register-home`, where users specify the home name and add multiple rooms. By default, each home includes an *Outdoor* room for devices monitoring external conditions like weather. After registration, home details, including device and rule configurations, are accessible at `http://client-url/home-details/homeId`, where `homeId` is the unique identifier of the home. This page also offers options to add devices, create rules (see Section 3.3.1), modify device values, and access the *Debugging Dashboard* (see Section 3.3.1).

The *Home and Device Manager* enables users to create virtual smart homes, modify device values, create rules, and access the *Debugging Dashboard*.

Devices can be added to the home by accessing the form at `http://client-url/add-device-to-room/homeId`. Users can then add multiple devices, configured using *Device Register* (see Section 3.3.2), into the rooms of the home. The added devices are listed under the Devices tab on the home details page, initially set to their default states from registration. The *Device_property* value of the devices can be changed using *Change Device Value* button, which opens a `Modal`. Users can select the *Device_property* from a `Dropdown`, view its current value, and input a new value in the provided field.

### Rule Manager

The *Rule Manager* allows the creation of TAP rules with a single trigger, multiple conditions, and a single action for a smart home.

The *Rule Manager* helps create TAP rules with a single trigger, multiple conditions, and a single action for a smart home. The form accessible at `http://client-url/create-rule/homeId` requires a name for each rule. Triggers are set by selecting a *Device_property*, a comparison operator (=, !=, <, >, <=, >=), and a trigger value. Actions are defined by selecting a *Device_property* and the desired new value. Conditions are formed using *AND*/*OR* groups and are implemented using React Query Builder[9]. The React Query Builder supports complex queries that match our need to build condition groups of TAP rules. Each condition can be set similarly to triggers. Created rules are listed under the Rules tab on the home details page.

## 3.3.2 Device Register

The *Device Register* allows users to add new devices with multiple sensors and actuators.

The *Device Register* page, accessible at `http://client-url/register-device`, provides a form for adding new devices. Users input the device name and can add multiple *Device_properties* (sensors and actuators). Each *Device_property* requires a name, state type (Fixed or Range), state values, and corresponding colors for the dashboard interface. For fixed states, users input concrete values and colors, with the first state as the default configured when the device is added to a smart home. Minimum and maximum values with colors are needed for ranged states, along with a default value. The application automatically adds a *Power State* actuator with fixed *On* (green) and *Off* (red) states, minimizing user input and ensuring at least one *Device_property* per device. On the *Device Register* page, users can view a list of configured devices beneath the device registration form. This list displays each device's name, its *Device_properties*, and their default values.

[9]https://react-querybuilder.js.org (last accessed on February 2, 2024)

## 3.4 Server-side Implementation

The server-side of the application is implemented using Flask[10] for building REST API and SQLite[11] database management, as described in Section 3.1. This combination helped develop this thesis's lightweight and complete backend application. Flask also offers future expansion of the application using extensions as needed. This section describes implementing two primary components: the REST API and the Database. The REST API provides `HTTP GET` and `POST` endpoints to support data exchange between the server and the client. The Database is used for storing persistent data and logs of the smart homes. The server endpoints are accessible at `http://server-url/` (e.g., `http://localhost:5001/` during the user study).

The server-side of the application is developed using Flask and SQLite to provide `HTTP` endpoints for data exchange and storage of persistent data.

The server implements the `check_rule` method to automate rule triggering in a smart home. This method is invoked whenever the state of a device's *Device_property* is updated. It takes the updated *Device_property* as a parameter and checks if this state change triggers any rules. If a rule is triggered, the method evaluates any configured conditions defined in the rule. Upon meeting these conditions, it updates the *Device_property* according to the action defined in the rule. Additionally, the method logs all actions performed during this process in the database, later visualized on the *Debugging Dashboard*.

`check_rule` method automates rule triggering in a smart home by evaluating triggers and conditions.

### 3.4.1 REST API

The REST API of the server offers two `HTTP` methods: `GET` and `POST`. `GET` requests retrieve data from the server. `POST` requests, on the other hand, require payload data to be sent to the server, used for submitting data to be processed and stored. The REST API uses Flask-SQLAlchemy[12], an ORM (Object Relational Mapper) extension for Flask,

The REST API offers `GET` and `POST` methods and communicates with the client via JSON.

[10]https://flask.palletsprojects.com (last accessed on February 2, 2024)
[11]https://www.sqlite.org/index.html (last accessed on February 2, 2024)
[12]https://flask-sqlalchemy.palletsprojects.com (last accessed on February 2, 2024)

to support interactions with the database. Communication between the server and client is handled using JSON (JavaScript Object Notation)[13] for data exchange. The implemented `GET` endpoints are described in Table 3.1, and the `POST` endpoints are described in Table 3.2. The endpoints are prefixed with `http://server-url/api/` (e.g., `http://localhost:5001/api/` during the user study).

### 3.4.2 Data Storage

The application uses SQLite as a database engine to store and organize persistent data.

Our application utilizes SQLite[14], a compact and efficient database engine to organize and store data. This choice aligns well with our Flask-based REST API server, supporting quick data operations and handling small data quantities of our thesis. Data is structured into entities for organized storage and easy retrieval. Table 3.3 describes these entities. We present the Entity-Relationship (ER) diagram, as shown in Figure 3.7, which visually represents the relationships between these data entities, providing a clear understanding of the system's data structure.

[13]https://www.json.org/json-en.html (last accessed on February 2, 2024)

[14]https://www.sqlite.org/index.html (last accessed on February 2, 2024)

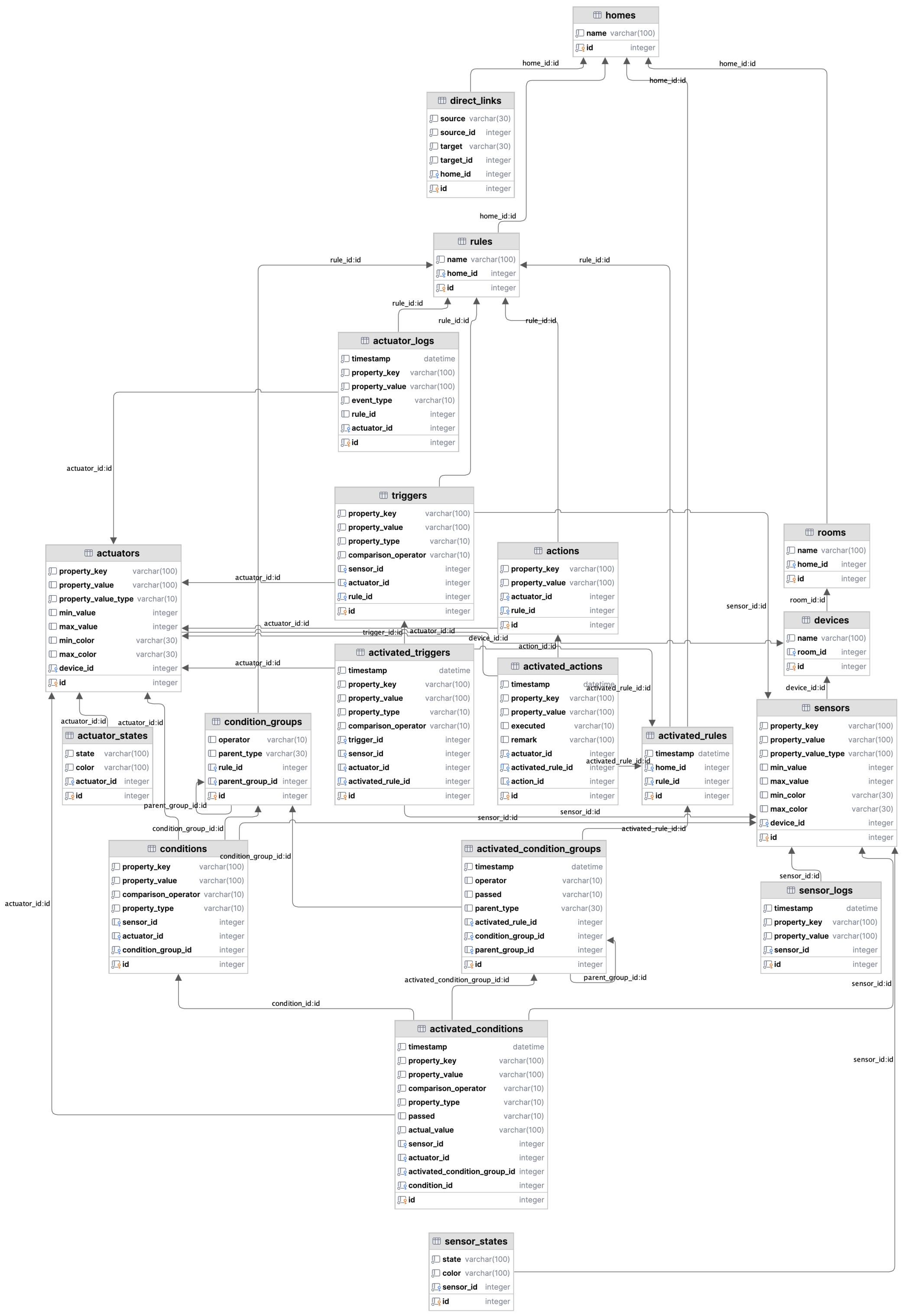


**Figure 3.7:** Entity-Relationship diagram of the database models - This diagram visually represents the structure and relationships of the database entities used in the smart home debugging web application.

| **Endpoint** | **Description** |
|---|---|
| `get-homes` | Returns a list of all homes configured in the application, including each home's ID, name, number of rooms, total devices across all rooms, and the number of associated rules. |
| `get-devices` | Returns a list of all devices registered by the *Device Register*, detailing each device's name, state type (Fixed or Range), value ranges, possible states, and colors. |
| `home-details/homeId` | Returns information about a specific smart home, including its name, associated rooms, rules, activated rules, and direct links. |
| `get-home-devices/homeId` | Returns details about sensors and actuators of devices in a specified smart home, including each device's name, state type, value ranges, and colors. |
| `unfiltered-trace-data/homeId` | Returns trace data for all activated rules and updates logs of sensors and actuators in a specific smart home. |
| `trace-data/homeId/ruleId` | Returns trace data for the specified activated rule in a smart home, including information about the rule, directly linked rules, and logs of involved sensors and actuators. |

**Table 3.1:** `HTTP GET` endpoints of the REST API server and the description of each endpoint's functionality and the type of data it retrieves. The endpoints are prefixed with `http://server-url/api/` (e.g., `http://localhost:5001/api/` during the user study).

| **Endpoint** | **Description** |
|---|---|
| `register-new-device` | Registers new smart devices. Requires data including the device's name, state type (Fixed or Range), value ranges, possible states, and associated colors. |
| `register-home` | Registers a new smart home in the application. Requires data containing the home's name and its rooms. |
| `add-device-to-room/homeId` | Assigns devices to specific rooms within a smart home. Needs data specifying which devices to add to each room in the home. |
| `update-device-value/homeId` | Allows updating the value of *Device_properties* for devices in a smart home. Requires data containing the device identifier and the new value to be set. |
| `create-rule/homeId` | Creates a new TAP rule in a specific smart home. It needs data, including the rule's name, trigger, and action details, and associated conditions. |

**Table 3.2:** `HTTP POST` endpoints of the REST API server and the description of each endpoint's purpose and the data it processes. The endpoints are prefixed with `http://server-url/api/` (e.g., `http://localhost:5001/api/` during the user study).

| Entity | Description |
|---|---|
| `Home` | Represents a virtual smart home entity. |
| `Room` | Represents a specific room within a Home. |
| `Device` | Represents a smart device, which can include Sensors and Actuators. |
| `Sensor` | Represents a sensor property of a Device. |
| `SensorState` | Stores the various fixed states or readings that a Sensor can support. |
| `SensorLog` | Logs historical state change data recorded by Sensors over time. |
| `Actuator` | Represents an actuator property of a Device. |
| `ActuatorState` | Stores the various fixed states or readings that an Actuator can support. |
| `ActuatorLog` | Logs historical state change data recorded by Actuators over time. |
| `Rule` | Represents a TAP rule, with a rule name. |
| `Trigger` | Specifies the event that triggers a Rule. |
| `ConditionGroup` | Groups multiple Conditions and ConditionGroups to determine a Rule's action execution. |
| `Condition` | Represents individual conditions within a Rule. |
| `Action` | Defines the action to be performed when a Rule is triggered and its conditions are met. |
| `ActivatedRule` | Tracks instances when a Rule is activated in the system. |
| `ActivatedTrigger` | Records the Trigger event of ActivatedRule. |
| `ActivatedConditionGroup` | Records the evaluation of ConditionGroups in an ActivatedRule. |
| `ActivatedCondition` | Records the evaluation of individual Conditions in an ActivatedRule. |
| `ActivatedAction` | Records the execution of Actions in ActivatedRule. |
| `DirectLink` | Represents a direct link between the ActivatedAction of an ActivatedRule and the ActivatedTrigger of a different ActivatedRule, which is activated because of the previous rule. |

**Table 3.3:** This table provides descriptions of entities used in the web application. Each entity is stored in a separate table in the database.

# Chapter 4

# Evaluation

In this chapter, we evaluate the *Debugging Dashboard* developed in Chapter 3, focusing on its effectiveness in identifying the cause of conflicts in TAP rules. To evaluate, we conducted a user study that compares the use of the dashboard with and without the data filtering feature. In Section 4.1, we postulate the hypotheses to be tested during the study. In Section 4.2, we describe the study designed to test the hypotheses, the procedure carried out during the study, and the participant demographics. Both quantitative and qualitative results are presented in Section 4.3. The chapter concludes with the evaluation discussion in Section 4.4.

## 4.1 Hypotheses

To answer our research questions outlined in Chapter 1, we postulate the following hypotheses to be tested during the user study:

**H1** The use of filtering and tracing relevant data in the dashboard increases the accuracy of identifying the causes of conflicts in TAP rules compared to not using data filtering.

**H1.1** Filtering and tracing relevant data in the dashboard makes identifying the cause of conflicts in TAP rules easier than not using data filtering.

**H1.2** Filtering and tracing relevant data in the dashboard reduces the time taken to identify the causes of conflicts in TAP rules compared to not using data filtering.

**H2** The use of filtering and tracing relevant data in the dashboard leads to higher confidence among users in identifying the causes of conflicts in TAP rules compared to not using data filtering.

**H2.1** Filtering and tracing relevant data in the dashboard improves users' understanding of conflicting situations in TAP rules compared to not using data filtering.

**H3** Users are more satisfied in identifying the cause of conflicts in TAP rules when filtering and tracing relevant data in the dashboard than when not using data filtering.

## 4.2 User Study

The user study evaluates the impact of filtering and tracing relevant data on debugging aspects.

The user study evaluates the *Debugging Dashboard* by comparing its use with filtering and tracing relevant data (the *Filter* condition) and without data filtering (the *Baseline* condition). The focus is on assessing the impact of filtering and tracing relevant data on various aspects of debugging, including accuracy, difficulty, speed, confidence, understanding, and satisfaction, in line with the hypotheses outlined in Section 4.1. This approach aims to gain insights into how filtering and tracing relevant data influences the TAP debugging process. We conducted a pilot study involving one participant to prepare for the user study. This resulted in minor design modifications for the main study, detailed in Section 4.2.1. The pilot study participant did not participate in the main study.

### 4.2.1 Study Design

A within-group study design was chosen for its effectiveness in controlling for individual differences among participants. This design allows each participant to experience both conditions (*Filter* and *Baseline*) of the study. Doing so reduces variability caused by differing skills, experiences, or preferences that individual participants may have. This approach enhances the reliability of the results, as comparisons are made within the same set of participants, giving a clearer view of the impact of the data filtering feature on their ability to debug TAP rule conflicts.

We chose a within-group study design to control for individual differences among participants and enhance the reliability of the results.

The study design involved four bug types in simulating common TAP rule conflicts, as identified in Table 2.1 by Coppers et al. [2022]. We selected *action inconsistent*, *action reversal*, *condition unsatisfiable*, and *infinite loop*. These particular bugs were chosen because they illustrate a range of issues in TAP systems, from logical inconsistencies to unsolvable conditions, and their potential to express them using visualization.

The study design involved four bug types in simulating common TAP rule conflicts.

The following eight debugging tasks were developed representing each bug type under two dashboard conditions (*Filter* and *Baseline*):

**AIB** *Action Inconsistent* using *Baseline* dashboard.

**AIF** *Action Inconsistent* using *Filter* dashboard.

**ARB** *Action Reversal* using *Baseline* dashboard.

**ARF** *Action Reversal* using *Filter* dashboard.

**CUB** *Condition Unsatisfiable* using *Baseline* dashboard.

**CUF** *Condition Unsatisfiable* using *Filter* dashboard.

**ILB** *Infinite Loop* using *Baseline* dashboard.

**ILF** *Infinite Loop* using *Filter* dashboard.

We created a set of conflicting rules for each task to represent the specific bug. To increase complexity and simulate

The study used a set of conflicting rules with additional noise rules executed in random order.

real-world scenarios, we included additional noise rules alongside the main rule set for each task, resulting in 5–8 rules and 8–11 *Device_properties* per task, aligning with the study design by Coppers et al. [2020]. These rules were executed in random order within separate virtual smart homes using the application to present each task in a realistic yet non-trivial manner. However, the semantics and execution order of the rules for each bug were identical under both dashboard conditions, differing only in room names and devices used in the rules. This maintained an identical number of rules and *Device_properties*, ensuring consistent cognitive difficulty for each bug type while introducing syntactic differences. We used Latin square to counterbalance the order of tasks presented to participants, mitigating any potential order effects. The study design materials can be found in Appendix A "User Study Design Materials".

An instruction page was created to introduce each task to the participants.

We created an instruction page to guide participants about the task to be performed for each task. The task number was displayed at the top of the page, followed by a table listing the configured rules in the virtual smart home. This table provided participants with an overview of the possible automation. Below the rules table, the task description outlined what the smart home user expected based on the rules, followed by a description of the unexpected situation. Participants were then instructed to identify the cause of the issue using the dashboard, accessible via a *Start* button at the bottom of the page.

The pilot study resulted in minor changes, including standardizing rule names, presenting rules in ascending order on the instruction page, and including task descriptions at the top of the dashboard page.

We conducted a pilot study with one participant (P0) at our lab to prepare and refine the study design for the main study. P0 did not participate in the main study. The feedback from the pilot study resulted in some minor changes. Clarification was added on the instruction page by labeling the *Filter* condition as *"With Filter"* and the *Baseline* condition as *"Without Filter"* to distinguish the available dashboard types more clearly. The rule names were standardized to *"Rule <rule_number>,"* with *<rule_number>* representing a random number. Rules were presented on the information page in ascending order for clarity. Additionally, to address the difficulty in remembering the issue and task description while using the dashboard, the task description was included at the top of the dashboard page, eliminating the need for participants to memorize it.

**Study Setup**

The user study was conducted in two locations: nine participants were in a quiet, closed room at our lab, while the remaining seven were in a quiet, closed room at the principal investigator's residence. In both locations, participants were provided with a desk and a chair. They performed the tasks on a 2020 M1 MacBook Air using a wired mouse. Participants were only allowed to use the mouse to interact with the dashboard for every task. The dashboard was accessed via Google Chrome in full-screen mode, with the MacBook's screen resolution set to 1680 by 1050.

### 4.2.2 Procedure

The user study involved several steps, including signing a consent form, completing a demographics questionnaire, a training task, and the main study.

The procedure for the user study involved several steps. Initially, participants were required to sign a consent form. Afterward, they completed a demographics questionnaire to gather background information, including their smart home, programming, and debugging experience. Next, they completed a training task with four rules and six *Device_properties*. This task was designed to familiarize them with TAP rules and the dashboard interface. Additionally, participants were introduced to the *Filter* feature of the dashboard. Once the dashboard's interface components were explained, the main study started. During the main study, participants were asked to think aloud to provide insight into their reasoning process. They could take breaks between tasks but not during them.

In the main study, participants first read the information page for each task, then clicked on the start button to access the dashboard to identify the cause of the issue. Time taken from starting the dashboard to informing the investigator of task completion was recorded without imposing any time limit. To evaluate the accuracy of the debugging dashboard, participants filled out a response sheet after each task. This sheet required them to indicate whether they identified the issue's cause and describe it. Furthermore, they evaluated their confidence, understanding, difficulty, and satisfaction on a 5-point Likert scale (Likert [1932]). For additional insights, participants were asked to propose potential solutions to the issues and describe their ideas.

In the end, participants filled out the System Causability Scale and provided feedback on their overall experience with the dashboard.

After completing all tasks, participants were requested to fill out the System Causability Scale (Holzinger et al. [2020]), an adaptation of the broad System Usability Scale (Brooke [1996]). This scale is specifically designed to evaluate the effectiveness of (visual) explanations, particularly in terms of their quality and clarity. Additionally, participants provided feedback on their overall experience with the dashboard. The forms used in the user study can be found in Appendix B "User Study Documents".

The study recorded the screen and audio and logged mouse events.

During the study, participants' screen and audio were recorded using OBS[1]. Consent for recording was obtained through the informed consent form. The recordings, stored anonymously, were deleted post-analysis. Mouse events during dashboard usage were logged and saved as CSV files via `HTTP POST` request to `http://server-url/save-mouse-log`, an endpoint used only for user studies. For data analysis, all collected data was digitized into CSV files and loaded into Jupyter[2] Notebooks. To analyze and visualize the data, we used Python libraries such as Pandas[3], SciPy[4], Matplotlib[5], and Seaborn[6].

### 4.2.3 Participants

We recruited sixteen participants (nine males, seven females) to complete our 8x8 Latin square design featuring eight tasks, repeated twice. The participants ranged in age from 24 to 60 years (mean = 35.31, median = 28, standard

[1]https://obsproject.com (last accessed on February 2, 2024)
[2]https://jupyter.org (last accessed on February 2, 2024)
[3]https://pandas.pydata.org (last accessed on February 2, 2024)
[4]https://scipy.org (last accessed on February 2, 2024)
[5]https://matplotlib.org (last accessed on February 2, 2024)
[6]https://seaborn.pydata.org (last accessed on February 2, 2024)

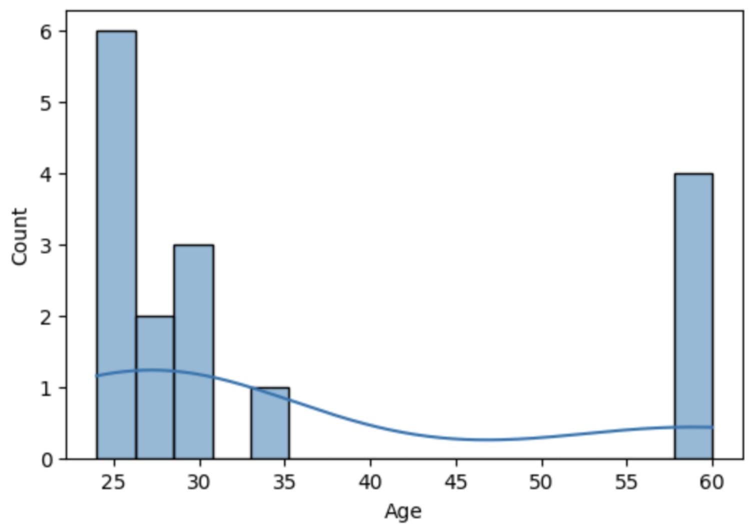


**Figure 4.1:** Histogram represents the age distribution of participants in the user study. The age range spans from 24 to 60 years, with a mean age of 35.31 years, a median age of 28 years, and a standard deviation of 14.36 years.

deviation = 14.36). They came from diverse backgrounds, including six from computer science, five from engineering, and one each from biology, rehabilitation engineering, floristry, design, and social work. Prior knowledge of TAP, coding, or debugging was optional for participation. Among the participants, only five owned smart home devices, and three had encountered undesired situations with their smart homes. Interestingly, one participant who did not own a smart home shared an experience of their neighbor being accidentally locked out while in the garden. Of the sixteen participants, thirteen, excluding the florist, designer, and social worker, had some programming experience from their studies or work. Six participants had over five years of programming experience, three had three to five years, and four had less than three years. Regarding debugging experience, five participants had none, three had less than a year, four had more than five years, two had three to five years, and two had one to two years.

Sixteen participants with diverse backgrounds and programming experience completed the study, with only five owning smart home devices.

## 4.3 Results

In this section, we will present and discuss both quantitative and qualitative results from our user study.

### 4.3.1 Quantitative Results

We analyzed the data from our user study to address our research questions. The analysis assessed the significance of data filtering in debugging accuracy, difficulty, speed, confidence, understanding, and satisfaction.

#### Debugging Accuracy

Data filtering did not improve identifying debugging accuracy in TAP rules, rejecting *H1*.

We assessed participants' accuracy in identifying the cause of TAP rule conflicts in both *Filter* and *Baseline* conditions. The analysis involved comparing the success rates between these two conditions to address *RQ1*. We used the Shapiro-Wilk test to determine the data's distribution and found that the success percentages were not normally distributed. Consequently, we employed the Wilcoxon Signed-Rank test, a non-parametric statistical method, for the significance analysis. This test showed no significant difference in accuracy between the *Filter* and *Baseline* conditions ($W = 20.50, p = 0.8028$), indicating that data filtering did not significantly improve participants' ability to identify the cause of conflict in TAP rules correctly. The mean success rates were 53.12% for *Filter* and 51.56% for *Baseline*. Based on these findings, *H1*, which suggested that filtering and tracing data improves accuracy in identifying conflicts, is rejected.

A learning effect was observed based on the task order given to participants, and different bug types had varying impacts on debugging accuracy.

The analysis revealed no significant difference in accuracy between the *Filter* and *Baseline* conditions. However, a learning effect was observed based on the task order given to participants, as shown in Figure 4.2. This learning effect was anticipated as participants' familiarity with the dashboard increased over multiple uses. Additionally, we noted variations in accuracy across different bug types, as shown in Figure 4.3, a finding not previously explored in the existing literature. To investigate further, we conducted the Friedman test, which demonstrated a significant impact of bug type on debugging accuracy ($\chi^2 = 11.19$, $p = 0.0107$). This suggests that the type of bug influences the success rate in identifying conflicts. The detailed results of the post-hoc pairwise comparison using the Wilcoxon Signed-Rank test are presented in Table 4.1.

| **Bug** | **Significance** | | **Mean Accuracy** |
|---|---|---|---|
| *Action Inconsistent* | A | | 62.50% |
| *Action Reversal* | A | | 62.50% |
| *Condition Unsatisfiable* | | B | 18.75% |
| *Infinite Loop* | A | | 65.62% |

**Table 4.1:** Significant differences in debugging accuracy per bug. Rows only show significant bugs. Significantly different bugs ($p <= 0.05$) are not connected by the same letter. The means are rounded to two decimal places.

The *condition unsatisfiable* bug was the most difficult to identify accurately, but data filtering did not affect accuracy for specific bug types.

The post-hoc analysis results clearly indicate that the *condition unsatisfiable* bug was significantly more challenging for participants to accurately identify the cause of the issue compared to other bugs. This raises the question of how data filtering influences accuracy for each specific bug type. To explore this, we also examined the significance in accuracy based on the task type, grouping data for each bug under both dashboard conditions (*Filter* and *Baseline*). The Friedman test revealed a significant impact of task type on accuracy ($\chi^2 = 24.26$, $p = 0.0010$). The post-hoc pairwise comparisons of tasks using the Wilcoxon Signed-Rank test, detailed in Table 4.2, showed that tasks involving the *condition unsatisfiable* bug were significantly more challenging for participants to identify the cause compared to other tasks accurately. However, the presence of data filtering did not significantly affect accuracy for each specific bug type, indicating that while certain types of bugs inherently present more difficulty, the debugging method (with or without data filtering) does not impact accuracy in these cases.

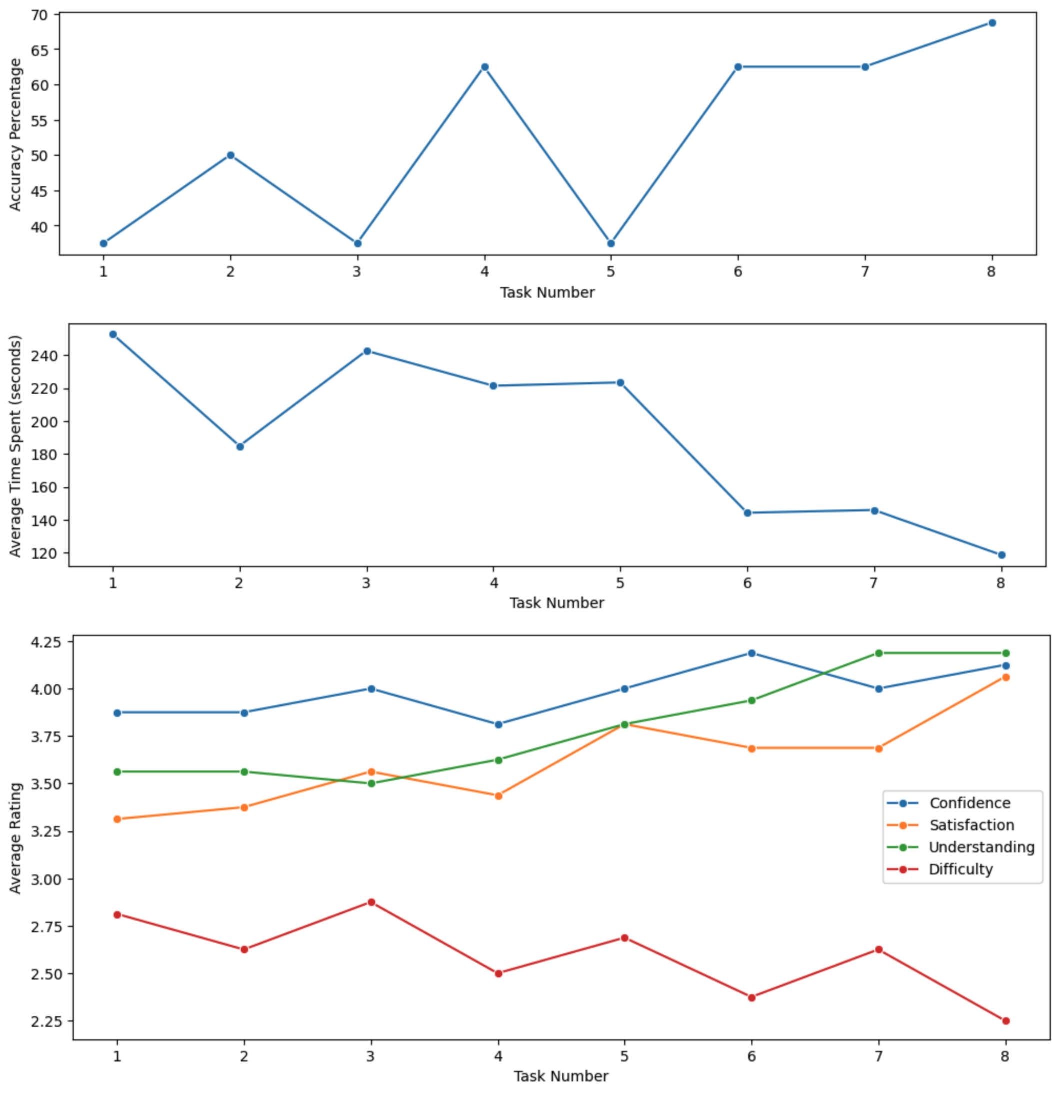


**Figure 4.2:** This line graph depicts the learning effect observed among participants during the user study. The x-axis represents the order of tasks performed by the participant. The y-axis represents the average accuracy percentage (top), average time spent in seconds (center), and average Likert scale ratings for confidence, satisfaction, understanding, and difficulty (bottom).

### Debugging Difficulty

The analysis for *RQ1.1* involved comparing participants' self-reported difficulty levels on a 5-point Likert scale in both *Filter* and *Baseline* conditions. The Shapiro-Wilk test

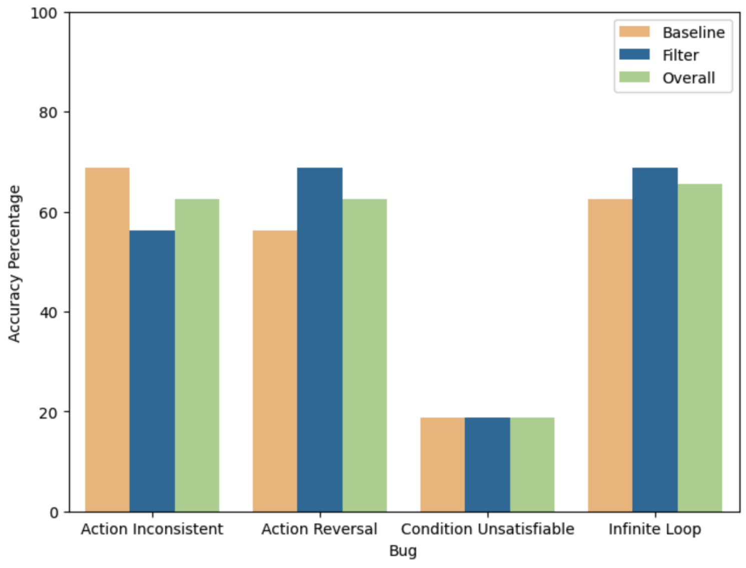


**Figure 4.3:** Average debugging accuracy percentage for Overall, *Filter*, and *Baseline* conditions per bug. The four bug types are shown on the x-axis, and the y-axis shows the average debugging accuracy percentage.

| Task | Significance | | Mean Accuracy |
|---|---|---|---|
| *AIB* | A | | 68.75% |
| *AIF* | A | | 56.25% |
| *ARB* | A | | 56.25% |
| *ARF* | A | | 68.75% |
| *CUB* | | B | 18.75% |
| *CUF* | | B | 18.75% |
| *ILB* | A | | 62.50% |
| *ILF* | A | | 68.75% |

**Table 4.2:** Significant differences in debugging accuracy per task. Rows only show significant tasks. Significantly different tasks ($p <= 0.05$) are not connected by the same letter. The means are rounded to two decimal places.

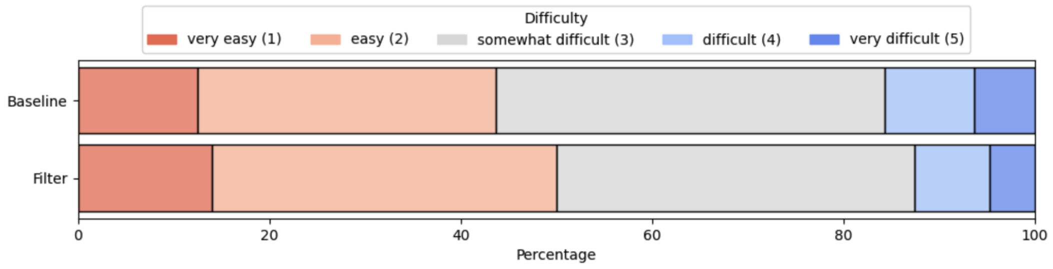


**Figure 4.4:** Distribution of debugging difficulty rating per task.

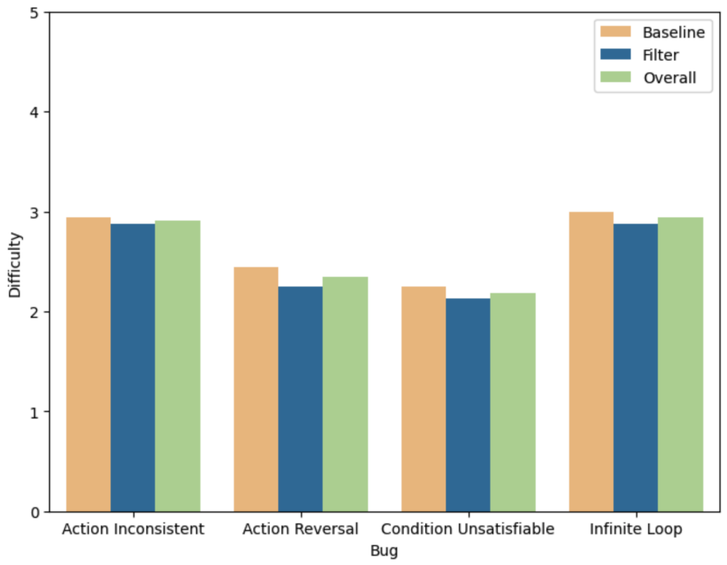


**Figure 4.5:** Average debugging difficulty for Overall, *Filter*, and *Baseline* conditions per bug. The four bug types are shown on the x-axis, and the y-axis shows the average debugging difficulty ratings.

Data filtering does not significantly impact the difficulty of identifying the cause of conflicts, rejecting *H1.1*.

indicated a non-normal distribution of difficulty levels. Thus, the Wilcoxon Signed-Rank test was used for significance testing. The results showed no significant difference in difficulty levels between the two conditions ($W = 38.00$, $p = 0.2050$). The average difficulty levels were 2.53 for *Filter* and 2.66 for *Baseline*, suggesting that data filtering does not significantly impact the difficulty of identifying the cause of conflicts, leading to the rejection of *H1.1*.

The analysis also examined the influence of different bug types on the perceived difficulty of identifying conflict causes. A Friedman test, assessing the impact of bug type on difficulty levels, showed significant results ($\chi^2 = 16.07$, $p = 0.0011$). This finding indicates that the type of bug significantly affects participants' difficulty levels. According to the post-hoc analysis using the Wilcoxon Signed-Rank test, detailed in Table 4.3, participants found the *condition unsatisfiable* bug to be significantly less difficult to identify compared to the *action inconsistent* and *infinite loop* bugs.

Bug type significantly affects difficulty levels, with *condition unsatisfiable* being less difficult to identify than *action inconsistent* and *infinite loop* bugs.

| Bug | Significance | | Mean | Std. Dev. |
|---|---|---|---|---|
| *Action Inconsistent* | A | | 2.91 | 1.02 |
| *Condition Unsatisfiable* | | B | 2.19 | 0.78 |
| *Infinite Loop* | A | | 2.94 | 0.95 |

**Table 4.3:** Significant differences in debugging difficulty per bug. Rows only show significant bugs. Significantly different bugs ($p <= 0.05$) are not connected by the same letter. The means and standard deviations are rounded to two decimal places.

We further explored the impact of task type on difficulty levels, categorizing data for each bug under the *Filter* and *Baseline* conditions. A significant impact of task type on difficulty levels was identified through the Friedman test ($\chi^2 = 22.73$, $p = 0.0019$), indicating variations in difficulty across different tasks. Post-hoc pairwise comparisons using the Wilcoxon Signed-Rank test, detailed in Table 4.4, indicated that while specific tasks were more challenging compared to other tasks, the presence of data filtering did not significantly affect difficulty levels for specific bug types, suggesting that the nature of the task influences difficulty more than the debugging method (with or without data filtering) used.

The result indicated variations in difficulty across different tasks, with specific tasks being more challenging than others.

### Debugging Speed

The analysis for *RQ1.2* focused on comparing debugging time in both *Filter* and *Baseline* conditions to assess the impact of data filtering on debugging speed. The Shapiro-

| Task | Significance | | Mean | Std. Dev. |
|---|---|---|---|---|
| *AIB* | A | | 2.94 | 0.85 |
| *AIF* | A | | 2.88 | 1.20 |
| *ARF* | | B | 2.25 | 0.77 |
| *CUB* | | B | 2.25 | 0.77 |
| *CUF* | | B | 2.12 | 0.81 |
| *ILB* | A | | 3.00 | 0.97 |
| *ILF* | A | | 2.88 | 0.96 |

**Table 4.4:** Significant differences in debugging difficulty per task. Rows only show significant tasks. Significantly different tasks ($p <= 0.05$) are not connected by the same letter. The means and standard deviations are rounded to two decimal places.

No significant difference was found in debugging time between *Filter* and *Baseline* conditions, leading to the rejection of *H1.2*.

Wilk test confirmed a non-normal distribution of debugging times. Thus, we used the Wilcoxon Signed-Rank test for significance testing. The results showed no significant difference in time taken between *Filter* and *Baseline* conditions ($W = 48.00$, $p = 0.4955$). The average debugging times were 195.39 seconds for *Filter* and 188.28 seconds for *Baseline*, indicating that data filtering does not significantly affect debugging speed, thus leading to the rejection of *H1.2*.

Bug type influenced debugging times significantly, with *infinite loop* taking longer and *condition unsatisfiable* taking less time.

The analysis additionally explored the impact of bug types on debugging speed using a Friedman test. Significant results ($\chi^2 = 11.48$, $p = 0.0094$) indicated that the type of bug significantly influenced participants' debugging times. The post-hoc analysis with the Wilcoxon Signed-Rank test, as detailed in Table 4.5, showed that participants took significantly more time to debug the *infinite loop* bug than other bugs. Conversely, they were significantly faster at debugging the *condition unsatisfiable* bug than the *action inconsistent* bug.

Task type has a significant impact on debugging speed, while the use of data filtering did not significantly affect the debugging speed for specific bug types.

We further examined the influence of task type on debugging speed, considering both the *Filter* and *Baseline* conditions. The Friedman test showed a significant impact of task type on debugging speed ($\chi^2 = 18.89$, $p = 0.0085$). The post-hoc pairwise comparisons, conducted using the Wilcoxon Signed-Rank test and detailed in Table 4.6, sug-

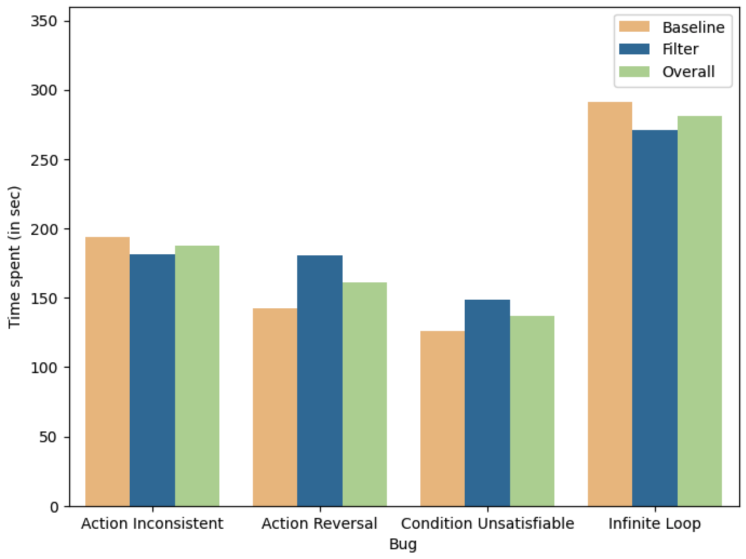


**Figure 4.6:** Average debugging speed for Overall, *Filter*, and *Baseline* conditions per bug. The four bug types are shown on the x-axis, and the y-axis shows the average debugging difficulty ratings.

gested that while specific tasks showed significant differences in debugging time, the use of data filtering did not significantly impact the speed for specific bug types. This finding implies that the nature of the task plays a more crucial role in influencing debugging speed than the debugging method (with or without data filtering) itself.

### User Confidence

To evaluate participants' confidence in identifying the causes of TAP rule conflicts, we compared their self-reported confidence levels using a 5-point Likert scale in both *Filter* and *Baseline* conditions. This analysis aimed to address *RQ2*. The Shapiro-Wilk test indicated a non-normal distribution, leading us to use the Wilcoxon Signed-Rank test for significance testing. The analysis showed no significant difference in confidence levels between the conditions ($W = 51.00$, $p = 0.6057$), with mean confidence lev-

No significant difference was found in user confidence levels in identifying TAP rule conflicts between the *Filter* and *Baseline* conditions, rejecting *H2*.

| Bug | Significance | | | Mean | Std. Dev. |
|---|---|---|---|---|---|
| *Action Inconsistent* | A | | | 187.44 | 120.60 |
| *Action Reversal* | A | B | | 161.47 | 102.82 |
| *Condition Unsatisfiable* | | B | | 137.19 | 78.47 |
| *Infinite Loop* | | | C | 281.25 | 160.92 |

**Table 4.5:** Significant differences in debugging speed per bug. Rows only show significant bugs. Significantly different bugs ($p <= 0.05$) are not connected by the same letter. The means and standard deviations are rounded to two decimal places.

| Task | Significance | | | | Mean | Std. Dev. |
|---|---|---|---|---|---|---|
| *AIB* | A | | | D | 193.75 | 99.40 |
| *ARB* | A | B | | | 142.12 | 100.92 |
| *ARF* | A | B | | D | 180.81 | 104.24 |
| *CUB* | | B | | | 126.06 | 65.42 |
| *CUF* | A | B | | | 148.31 | 90.45 |
| *ILB* | | | C | | 291.19 | 163.49 |
| *ILF* | | | C | D | 273.31 | 163.03 |

**Table 4.6:** Significant differences in debugging speed per task. Rows only show significant tasks. Significantly different tasks ($p <= 0.05$) are not connected by the same letter. The means and standard deviations are rounded to two decimal places.

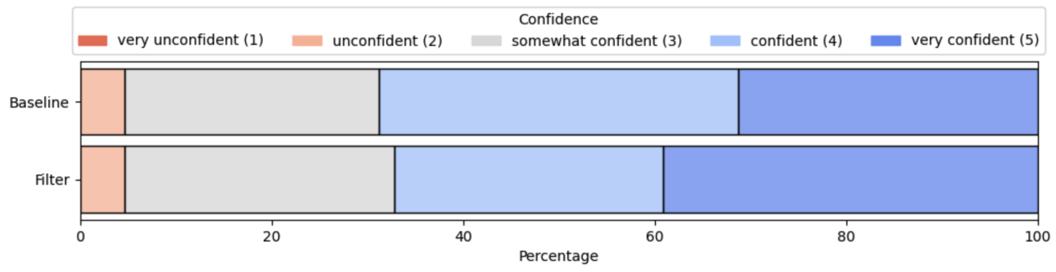


**Figure 4.7:** Distribution of debugging difficulty rating per task.

els at 4.02 for *Filter* and 3.95 for *Baseline*. These findings suggest that data filtering does not significantly impact user confidence in identifying conflicts, leading to rejecting *H2*.

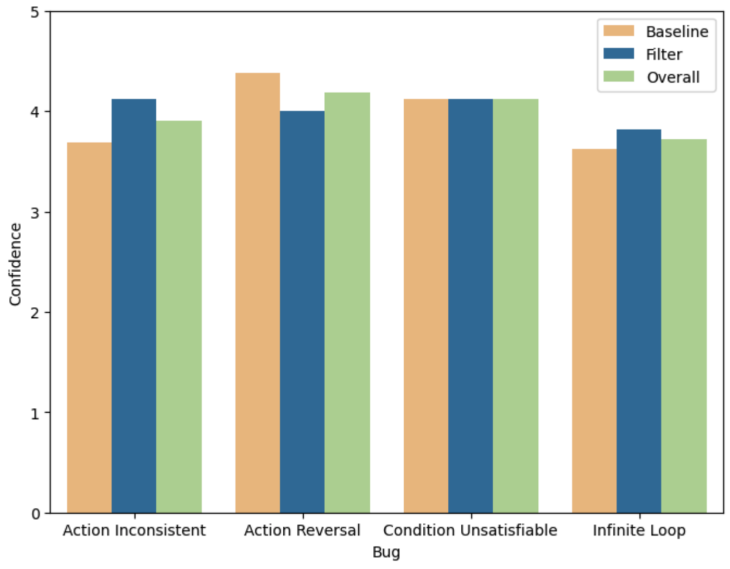


**Figure 4.8:** Average user confidence for Overall, *Filter*, and *Baseline* conditions per bug. The four bug types are shown on the x-axis, and the y-axis shows the average debugging difficulty ratings.

To understand further the influence of bug types on participants' confidence levels, we analyzed the data, as depicted in Figure 4.8. A Friedman test was conducted to assess the impact of bug type on confidence levels, which showed significant results ($\chi^2 = 9.98, p = 0.0188$). The result indicates that the type of bug significantly affects participants' confidence in identifying the cause of conflicts. The post-hoc analysis using the Wilcoxon Signed-Rank test, as detailed in Table 4.7, showed participants exhibited significantly lower confidence when identifying the cause of the *infinite loop* bug. We also investigated the influence of task type on confidence levels, grouping data for each bug under both *Filter* and *Baseline* conditions. However, the Friedman test indicated no significant impact of task type on confidence levels ($\chi^2 = 12.40, p = 0.0882$), suggesting that the type of task (with or without data filtering) did not significantly affect participants' confidence levels.

We found that the type of bug significantly affects participants' confidence in identifying the cause of conflicts, with the *infinite loop* bug resulting in lower confidence levels.

| Bug | Significance | Mean | Std. Dev. |
|---|---|---|---|
| *Action Reversal* | A | 4.19 | 0.78 |
| *Condition Unsatisfiable* | A | 4.12 | 0.83 |
| *Infinite Loop* | B | 3.72 | 0.73 |

**Table 4.7:** Significant differences in user confidence per bug. Rows only show significant bugs. Significantly different bugs ($p <= 0.05$) are not connected by the same letter. The means and standard deviations are rounded to two decimal places.

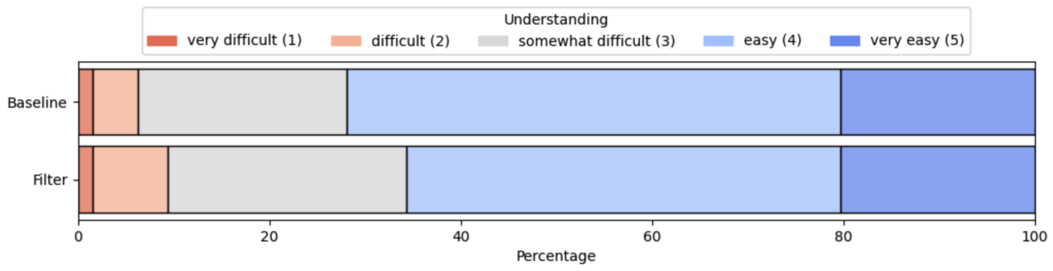


**Figure 4.9:** Distribution of conflict understanding rating per task.

## Conflict Understanding

Data filtering does not significantly affect understanding TAP rule conflicts, rejecting *H2.1*.

In evaluating participants' understanding of TAP rule conflicts, we compared their self-reported understanding on a 5-point Likert scale in both *Filter* and *Baseline* conditions. The analysis, aimed at addressing *RQ2.1*, used the Shapiro-Wilk test to confirm non-normal distribution and employed the Wilcoxon Signed-Rank test for significance testing. The result revealed no significant difference in understanding levels between the two conditions ($W = 31.00$, $p = 0.5296$), with mean understanding levels of 3.75 for *Filter* and 3.84 for *Baseline*. This outcome indicates that data filtering does not significantly affect understanding TAP rule conflicts. Consequently, *H2.1* is rejected based on these results.

We further analyzed the influence of bug types on participants' understanding levels, as shown in Figure 4.10. The Friedman test indicated a significant impact of bug type on understanding levels ($\chi^2 = 7.98$, $p = 0.0465$), suggesting bug type significantly affects understanding of TAP

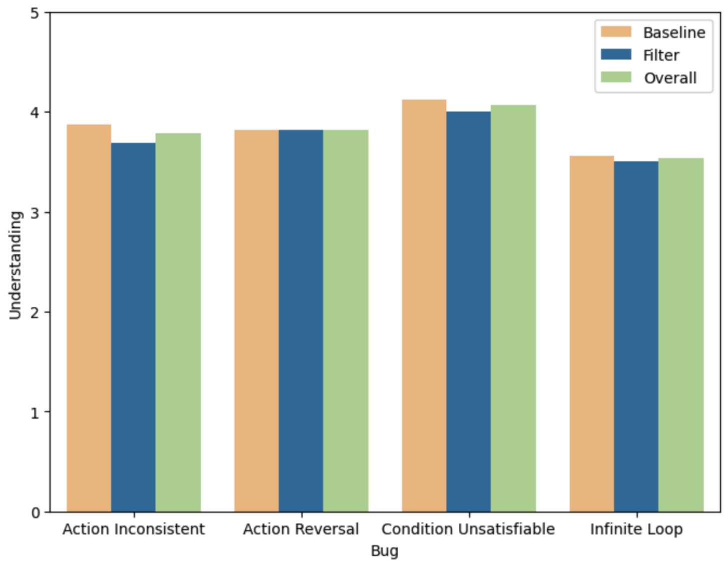


**Figure 4.10:** Average conflict understanding for Overall, *Filter*, and *Baseline* conditions per bug. The four bug types are shown on the x-axis, and the y-axis shows the average debugging difficulty ratings.

rule conflicts. The post-hoc analysis, using the Wilcoxon Signed-Rank test and detailed in Table 4.8, revealed a significantly better understanding of the *condition unsatisfiable* bug compared to the *infinite loop* bug. Further investigation into the influence of task type on understanding levels, however, showed no significant impact ($\chi^2 = 11.82$, $p = 0.1067$), indicating that the method of debugging (with or without data filtering) did not significantly affect understanding levels.

Bug type significantly affects the understanding of TAP rule conflicts, with *condition unsatisfiable* bug having a better understanding than *infinite loop* bug.

| Bug | Significance | | Mean | Std. Dev. |
|---|---|---|---|---|
| *Condition Unsatisfiable* | A | | 4.12 | 0.76 |
| *Infinite Loop* | | B | 3.72 | 0.84 |

**Table 4.8:** Significant differences in conflict understanding per bug. Rows only show significant bugs. Significantly different bugs ($p <= 0.05$) are not connected by the same letter. The means and standard deviations are rounded to two decimal places.

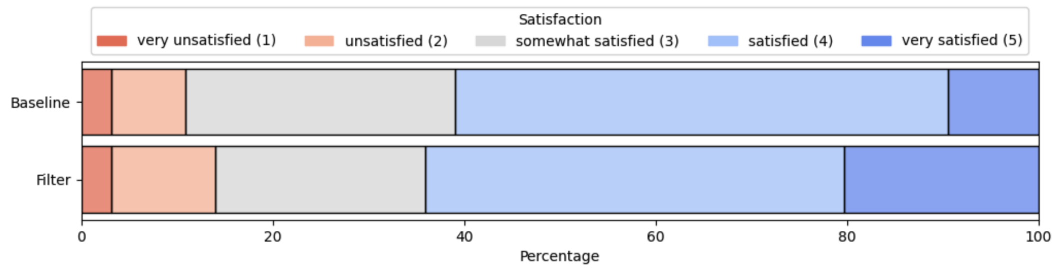


**Figure 4.11:** Distribution of user satisfaction rating per task.

### User Satisfaction

Data filtering does not significantly impact satisfaction in identifying the cause of conflicts, rejecting *H3*.

To address *RQ3*, we analyzed participants' self-reported satisfaction levels using a 5-point Likert scale in both *Filter* and *Baseline* conditions. We found that the satisfaction levels were not normally distributed using the Shapiro-Wilk test. We used the Wilcoxon Signed-Rank test for significance testing. Our analysis revealed no significant difference in satisfaction levels between the two conditions ($W = 33.50$, $p = 0.3981$). The mean satisfaction levels were 3.67 for *Filter* and 3.56 for *Baseline*. These findings suggest that data filtering does not significantly impact satisfaction in identifying the cause of conflicts, leading to the rejection of *H3*.

Bug type significantly affects participants' satisfaction in identifying conflict causes, and participants were significantly more satisfied when identifying the cause of the *condition unsatisfiable* bug compared to other bug types.

The analysis further investigated the impact of bug type on satisfaction levels. The mean satisfaction levels based on bug types are shown in Figure 4.12. A Friedman test assessing this impact showed significant results ($\chi^2 = 14.18$, $p = 0.0027$), suggesting that bug type significantly affects participants' satisfaction in identifying conflict causes. Post-hoc analysis using the Wilcoxon Signed-Rank test, detailed in Table 4.9, revealed that participants were significantly more satisfied when identifying the cause of the *condition unsatisfiable* bug compared to other bug types. The results highlight the influence of specific bug characteristics on user satisfaction during the debugging process.

We further explored the influence of task type on satisfaction levels by grouping data for each bug under both the *Filter* and *Baseline* conditions. The Friedman test revealed a significant impact of task type on satisfaction levels ($\chi^2 =$

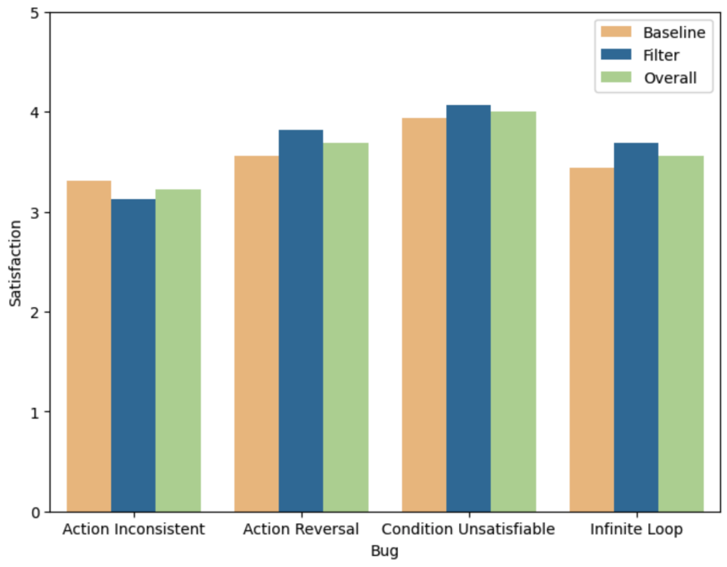


**Figure 4.12:** Average user satisfaction for Overall, *Filter*, and *Baseline* conditions per bug. The four bug types are shown on the x-axis, and the y-axis shows the average debugging difficulty ratings.

$17.64, p = 0.0137$), indicating that specific tasks were significantly more satisfactory for participants in identifying the cause compared to others. However, the post-hoc pairwise comparisons of tasks using the Wilcoxon Signed-Rank test show that data filtering did not significantly affect user satisfaction for each specific bug type, as detailed in Table 4.10. The analysis suggests that while the nature of the task influences satisfaction, data filtering does not have a considerable effect on this aspect.

The influence of task type on satisfaction levels indicated a significant impact, but data filtering did not significantly affect user satisfaction for each bug type.

### System Causability Scale

We used the System Causability Scale (SCS) (Holzinger et al. [2020]) to evaluate the quality of both the *Filter* and *Baseline* dashboards in explaining the causes of TAP rule conflicts. The *Filter* dashboard scored 0.79, while the *Baseline* dashboard scored 0.72. The rating distribution was normal, as confirmed by the Shapiro-Wilk test. A Paired

SCS indicated that the *Filter* dashboard had a better quality of explanation for TAP rule conflicts than the *Baseline* dashboard.

| Bug | Significance | | Mean | Std. Dev. |
|---|---|---|---|---|
| *Action Inconsistent* | A | | 3.22 | 1.10 |
| *Condition Unsatisfiable* | | B | 4.00 | 0.80 |
| *Infinite Loop* | A | | 3.56 | 0.62 |

**Table 4.9:** Significant differences in user satisfaction per bug. Rows only show significant bugs. Significantly different bugs ($p <= 0.05$) are not connected by the same letter. The means and standard deviations are rounded to two decimal places.

| Task | Significance | | | | Mean | Std. Dev. |
|---|---|---|---|---|---|---|
| *AIB* | | | C | D | 3.31 | 0.70 |
| *AIF* | | | | D | 3.12 | 1.41 |
| *ARF* | A | B | C | | 3.81 | 0.91 |
| *CUB* | | B | | | 3.94 | 0.68 |
| *CUF* | A | B | | | 4.06 | 0.93 |
| *ILB* | A | | C | D | 3.44 | 0.73 |

**Table 4.10:** Significant differences in user satisfaction per task. Rows only show significant tasks. Significantly different tasks ($p <= 0.05$) are not connected by the same letter. The means and standard deviations are rounded to two decimal places.

t-test showed a significant difference between the *Filter* ($M = 3.90, SD = 0.60$); with $t(15) = -5.06, p < 0.0001$ and the *Baseline* ($M = 3.61, SD = 0.55$). The statistically significantly higher rating for the *Filter* dashboard than the *Baseline* dashboard indicates that the participants found the quality of explanation on the *Filter* dashboard better than the *Baseline* version.

The analysis of the SCS ratings for each question, as detailed in Table 4.11, showed that the ratings were not normally distributed, confirmed by the Shapiro-Wilk test. Therefore, the Wilcoxon Signed-Rank test was used for significance testing. The results indicated that participants rated the *Filter* dashboard higher than the *Baseline* dashboard for the statements *"I could change the level of detail on demand"* ($W = 0.0$, $p = 0.0008$) and *"I did not need more*

| Question | *Baseline* Rating | *Filter* Rating |
|---|---|---|
| 01. Factors in data | 3.94 | 4.06 |
| 02. Understood | 4.25 | 4.13 |
| 03. Change detail level | 2.25 | 4.19 |
| 04. Don't Need support | 3.50 | 3.63 |
| 05. Understanding causality | 3.94 | 4.25 |
| 06. Use with knowledge | 3.81 | 3.81 |
| 07. No inconsistencies | 4.06 | 4.18 |
| 08. Learn to understand | 2.94 | 3.25 |
| 09. Don't Need references | 3.25 | 3.69 |
| 10. Efficient | 4.13 | 4.13 |
| $SCS = \sum_i Rating_i/50$ | 0.72 | 0.79 |

**Table 4.11:** The table reports the System Causability Scale (SCS) ratings for *Baseline* and *Filter* versions of the dashboard.

*references in the explanations"* ($W = 0.0, p = 0.0384$).

### 4.3.2 Qualitative Results

In this section, we present the qualitative results of our study. We discuss participant feedback on the *Filter* dashboard and other dashboard components, providing insights into user experiences and perceptions of the dashboard's usability and effectiveness in debugging TAP rule conflicts.

#### *Filter* Feature

Six participants found the *Filter* helpful for clarifying causality and simplifying the debugging process.

Out of sixteen participants, six found the *Filter* feature particularly useful. P01 and P10 appreciated its ability to clarify causality and simplify the debugging process, with P01 stating it *"helped to determine causality"* and P10 stating it *"made it easier to find the reason."* P02 and P16 valued the feature for its data-filtering capability, with P02 liking how it *"filters out excess event information"* and P16 finding the reduced information *"convenient."* P08 praised the *Filter* feature for significantly easing problem identification, describing it as *"very easy to identify the problem & very useful."*

Two participants did not use the *Filter* feature, stating that the *Unfiltered View* provided sufficient information for them.

Surprisingly, two participants (P06 and P07) chose not to use the *Filter* feature. P06, with an engineering background, felt that the *Unfiltered View* of the dashboard provided sufficient information without the need for filtering data. They stated, *"the information on the dashboard is enough for me, I don't need to see isolated data...I didn't see any need for it."* P06 successfully solved 3 out of 4 tasks in the *Baseline* condition and 2 out of 4 in the *Filter* condition. Similarly, P07, from a computer science background, believed they could understand the rules without the filter feature, saying, *"I was able to understand the rules without filter, so I didn't need it."* P07 accurately solved 3 out of 4 tasks in both *Baseline* and *Filter* conditions.

Two participants used the *Filter* feature less and yet solved tasks successfully, indicating that the *Unfiltered View* was sufficient for debugging.

Participant P09, a computer science student, used the *Filter* feature for only one task (*ILF*) out of the four tasks in the *Filter* condition. In a post-study interview, they said, *"I utilized in the beginning as everything was new. Once I got used to the interface, I was able to filter out without filter feature."* P09 successfully solved three out of the four tasks in both *Baseline* and *Filter* conditions. Similarly, participant P11, a biologist who used the *Filter* feature for only two tasks (*ARF* and *ILF*) in the *Filter* condition, remarked, *"The filter was not really useful for me. Seeing it in a timeline was enough."* The feedback suggests that for P11, the standard *Unfiltered View* in the *Filter* condition provided sufficient information for debugging. Notably, P11 successfully solved three out of four tasks in the *Baseline* condition and all four tasks in the *Filter* condition, indicating a high level of proficiency in identifying the cause of TAP rule conflicts regardless of the dashboard version.

### Rule Timeline

Participants had mixed opinions on the *Rule Timeline* aspect of the dashboard, with some finding it helpful and others confusing.

In the *Rule Timeline* of the dashboard, which primarily focused on the causal dependencies of TAP rules, participants expressed diverse opinions. P04, P05, P06, P10, and P15 found the *Rule Timeline* aspect of the dashboard to be particularly useful, with P06 specifically appreciating the visual representation of causal dependencies between triggers and actions. However, P03 expressed a contrasting view, finding the *Rule Timeline* aspect confusing.

### Device Timeline

The *Device Timeline* feature of the dashboard received varied feedback from participants. Many found it helpful, with comments focusing on its clarity in understanding device status changes and their triggers. P03, P10, P11, P12, and P14 all mentioned using the device status bars to gain insights into the smart home's functioning. P10 specifically noted how it helped in identifying triggers. However, P04 reported not using the *Device Timeline* of the dashboard.

Participants had mixed feedback on the *Device Timeline* feature; some found it helpful, while one did not use it.

### Flash Icon

The flash icon feature of the dashboard was well-received for its ability to clarify whether events were triggered manually or automatically. Participants P05 and P15 appreciated the information the icon provided about the nature of events. Participant P09 found it particularly helpful, stating that flash icon *"helped me identify if an event was manually or automatically triggered. Logically, you can figure it out, but visually, it helps a lot. I would have given up in some cases if not for that."*

The flash icon clarified manual versus automatic event triggers and was found helpful by participants.

### Arrows

The arrows in the dashboard, which illustrate the chain of events triggered by TAP rules, received positive feedback for its use in the debugging process. Participant P09 specifically mentioned the usefulness of these arrows in quickly identifying the sequence of triggered events. This feature allowed for an easier and more efficient understanding of the causality within the smart home system. Similarly, participant P14 appreciated the arrows for clearly showing the

Participants found the arrows helpful in identifying the sequence of events and understanding causality in TAP rules.

chain of triggering events, enhancing their ability to follow the flow and impact of actions within the system.

### Condition Status Indicator

Participants appreciated the usefulness of the *Condition Status Indicator* for clarifying the status of conditions and simplifying the debugging process.

The *Condition Status Indicator* of the dashboard was appreciated by participants for its utility in clarifying the status of conditions within TAP rules. Participant P11 valued the ability to quickly identify which conditions were met and which were not, highlighting the feature's role in simplifying the debugging process. Similarly, Participant P14 found the *Condition Status Indicator* particularly helpful, providing clear and direct feedback on the state of rule conditions.

## 4.4 Discussion

The *Filter* version of the dashboard had better explanation quality but no significant difference in debugging aspects compared to the *Baseline* version.

Our research has revealed that the *Filter* version of the dashboard received a significantly higher rating on the System Causability Scale (SCS) than the *Baseline* version. This implies that the *Filter* version provided a better quality of explanation compared to the *Baseline* version. However, no significant difference was observed between the two versions regarding debugging accuracy, difficulty, speed, confidence, understanding, and satisfaction. This indicates that even though users preferred the *Filter* version, it did not provide any additional capabilities or insights that could improve the debugging of TAP conflicts. The feedback from participants P06, P07, P09, and P11 also supported this. Figure 4.13 presents further analysis of the amount of time spent by users on the *Filtered View* during the *Filter* condition. The data confirms that most participants spent little time on the *Filtered View* and mostly used the dashboard without filtering data.

Additionally, we observed a learning curve in terms of debugging accuracy, difficulty, speed, confidence, understanding, and satisfaction associated with the dashboard, as shown in Figure 4.2. This suggests that while participants appreciated the potential of the *Filter* version, within

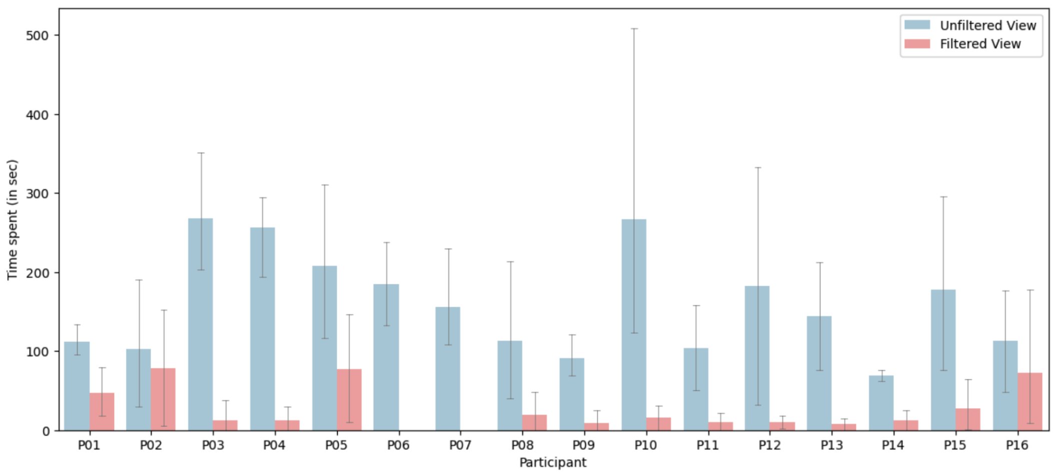


**Figure 4.13:** Time spent on *Filtered View* versus *Unfiltered View* during the *Filter* condition task per participant. The x-axis shows each participant. The y-axis shows the time spent in seconds.

the confines of the study, they might not have fully mastered its use, leading to similar results as the *Baseline* version.

The study showed significant differences in debugging based on bug type.

Although our study showed no significant differences in debugging between the *Baseline* and *Filter* versions, it highlighted significant differences in outcomes based on bug types. This discovery highlights the critical impact of a bug's nature on the debugging process, suggesting that different types of bugs demand varied cognitive abilities for effective conflict resolution. This finding is particularly insightful as it brings to light an aspect of TAP bugs, to the best of our knowledge, not explored in existing research, opening a new direction for understanding the complexities inherent in debugging different bug types.

Participants were accurate with *infinite loop* but less so with *condition unsatisfiable*; however, they were confident and satisfied in identifying the latter despite it being more complex.

After analyzing the results based on bug type, it was found that participants were most accurate in identifying the cause of the *infinite loop* bug and least accurate in identifying the cause of the *condition unsatisfiable* bug. Surprisingly, participants reported higher confidence, satisfaction, and understanding and less difficulty identifying the cause of the *condition unsatisfiable* bug compared to the *infinite loop* bug. Additionally, participants were fastest in completing *condition unsatisfiable* bug tasks and slowest in completing the *infinite loop* task. The results indicate that even though some bugs in TAP rule systems are inherently more complex to solve, this complexity does not necessarily reduce the confidence or satisfaction of the participants working to resolve these issues. This means that participants may feel confident and satisfied in their approach to solving a complex problem, regardless of the actual difficulty of the problem itself.

# Chapter 5

# Summary and Future Work

To conclude this thesis, we will summarize our research and contributions to smart home debugging. Additionally, we will discuss the potential limitations of our work and suggest directions for future research to build upon our findings.

## 5.1 Summary and Contributions

The thesis explored trace-based debugging for resolving conflicting TAP rules.

Our thesis investigated the potential of trace-based debugging to debug conflicting TAP rules. To begin with, we conducted a comprehensive review of previous research in this area, including the evolution of TAP. Subsequently, we examined various undesirable conflicts in TAP rules, their types, and definitions and explored different solutions proposed in existing research using static and dynamic analysis approaches.

We developed and evaluated a visual trace debugging dashboard to identify TAP rule conflicts.

Our work focused on developing and evaluating a visual trace debugging dashboard that helps identify the root causes of conflicts in TAP rules. We provided an overview of the web application incorporating the dashboard. Then, we discussed the *Debugging Dashboard*'s interface, design choices, and the application and dashboard implementation in detail. We evaluated two versions of the dashboard: the *Baseline* version without data filtering feature and the *Filter* version with data filtering feature.

The study found no significant differences between *Baseline* and *Filter* versions despite higher SCS ratings for the *Filter* version.

The evaluation involved a user study with 16 participants, assessing the dashboard's impact on various aspects of debugging, such as accuracy, difficulty, speed, confidence, understanding, and satisfaction. Interestingly, despite higher System Causability Scale ratings for the *Filter* version, no significant differences were observed between the *Baseline* and *Filter* versions across debugging parameters. This result suggests that while users appreciated the potential of the *Filter* version, they did not fully utilize its capabilities within the study's context.

The study suggests that different bugs require different cognitive strategies for resolution, indicating a need for further research.

One of the key findings of the study was that user performance varied significantly depending on the type of bug encountered, indicating that different bugs require different cognitive strategies for resolution. This insight opens new avenues for future research, particularly in enhancing user interaction with debugging tools and developing more effective approaches for handling diverse bug types in TAP systems.

## 5.2 Limitations and Future Work

This thesis provides an evaluation of a visual debugging dashboard for TAP rules. However, the study had some limitations that suggest potential opportunities for future research.

Participants underutilized the *Filter* feature, highlighting a need for more intuitive and engaging ways to present filtered data.

One of the limitations of the study was that while the *Filter* feature was innovative, not all participants utilized it fully. Future development could focus on making this feature more intuitive and engaging. Investigating different ways to present filtered data could significantly enhance interaction and understanding.

Another limitation was that the study focused only on four bug types. Our study revealed significant differences in debugging performance based on the type of bug encountered. Future research could broaden this scope by incorporating a diverse array of bugs. This expansion could involve simulations of complex smart home environments, including more complex scenarios that reflect real-world applications.

Further research on diverse TAP bugs could reveal significant differences.

Moreover, the study's participant pool was relatively small and might not represent the diverse demographic of TAP system users. Future studies could target a wider audience, including users from various expertise backgrounds.

Future studies can involve a large participant pool with various backgrounds.

Lastly, the constrained interaction time with the dashboard in this study might have prevented participants from fully adapting to its features. Subsequent studies could extend these interaction periods or include detailed training sessions. Such an approach could use longitudinal studies or interactive workshops, giving users ample time to familiarize themselves with the dashboard's functionalities and providing richer data on long-term user engagement and learning curves.

Extending interaction time and training sessions in subsequent studies could provide richer data on long-term user engagement and learning curves for the dashboard.

Each of these suggestions presents unique challenges but also offers the potential for significant advancements in understanding and improving user interactions with TAP systems and their debugging processes.

# Appendix A

# User Study Design Materials

Table A.1 shows an 8x8 Latin square design, replicated twice, to assign tasks to all 16 participants of our user study.

| **Task 1** | **Task 2** | **Task 3** | **Task 4** | **Task 5** | **Task 6** | **Task 7** | **Task 8** |
|---|---|---|---|---|---|---|---|
| ILB | ILF | AIB | ARF | CUB | ARB | CUF | AIF |
| ILF | ARF | ILB | ARB | AIB | AIF | CUB | CUF |
| ARF | ARB | ILF | AIF | ILB | CUF | AIB | CUB |
| ARB | AIF | ARF | CUF | ILF | CUB | ILB | AIB |
| AIF | CUF | ARB | CUB | ARF | AIB | ILF | ILB |
| CUF | CUB | AIF | AIB | ARB | ILB | ARF | ILF |
| CUB | AIB | CUF | ILB | AIF | ILF | ARB | ARF |
| AIB | ILB | CUB | ILF | CUF | ARF | AIF | ARB |

**Table A.1:** An 8x8 Latin square illustrating the balanced distribution of tasks.

Figures A.1, A.2, A.3, and A.4 present the rule sets designed for the four bug types used in our user study. Each bug type includes two sets of rules, one each for *Baseline* and *Filter* tasks.

Figures A.5 and A.6 show the instruction page for *Baseline* and *Filter* tasks, respectively.

Action Inconsistent TAP Rules

| Rule Description | Set 1 (AIB) | Set 2 (AIF) |
|---|---|---|
| If (A), while (X) holds then do (B) | Rule 2 | Rule 3 |
| If (C), while (Y) holds then do (D) | Rule 4 | Rule 5 |
| If (E), while (X AND Y) holds then do (P) | Rule 5 | Rule 1 |
| If (Y) then do (Not E) | Rule 1 | Rule 2 |
| If (F), while (E) holds then do (Q) | Rule 3 | Rule 4 |

Action Inconsistent Variables

| Variables | Set 1 (AIB) | Set 2 (AIF) |
|---|---|---|
| A | Outdoor Thermostat temperature > 27°C | Outdoor brightness < 30% |
| B | Bedroom AC temperature = 25°C | Living room Light brightness = 50% |
| C | Outdoor Thermostat temperature > 22°C | Outdoor brightness < 70% |
| D | Bedroom AC temperature = 20°C | Living room Light brightness = 90% |
| E | Bedroom Light On | Living room AC On |
| F | Outdoor brightness < 40% | Outdoor thermostat temperature > 27°C |
| X | Bedroom Window Close | Living room Curtains Close |
| Y | Bedroom Door Close | Living room TV Off |
| P | Bedroom AC On | Living room Window Close |
| Q | Bedroom Light brightness = 70% | Living room AC temperature = 23°C |

Action Inconsistent Rule Execution Order

| Set 1 (AIB) | Set 2 (AIF) |
|---|---|
| Rule 1 | Rule 2 |
| Rule 5 | Rule 1 |
| Rule 4 | Rule 5 |
| Rule 3 | Rule 4 |
| Rule 2 | Rule 3 |
| Rule 4 | Rule 5 |

**Figure A.1:** Rule sets for *Action Inconsistent* bug. Set 1 is used for the *Baseline* version and Set 2 for *Filter* version of the dashboard. Conflicting rules are highlighted.

Action Reversal TAP Rules

| Rule Description | Set 1 (ARB) | Set 2 (ARF) |
|---|---|---|
| If (A), while (E AND F) holds then do (B) | Rule 3 | Rule 2 |
| If (B), while (E AND F) holds then do (C) | Rule 6 | Rule 5 |
| If (C), while (E) holds then do (NOT A) | Rule 1 | Rule 3 |
| If (G), then do (H) | Rule 4 | Rule 6 |
| If (F), while (E) holds then do (I) | Rule 2 | Rule 1 |
| If (H), while (J OR K) holds then do (L) | Rule 5 | Rule 4 |

Action Reversal Variables

| Variables | Set 1 (ARB) | Set 2 (ARF) |
|---|---|---|
| A | Kitchen Light On | Living Room TV On |
| B | Bedroom Light Off | Living room Curtains Close |
| C | Living room Light On | Living room Light brightness = 90% |
| E | Outdoor Light brightness > 70% | Living room Light On |
| F | Bedroom Door Close | Outdoor Light brightness < 30% |
| G | Living room TV On | Kitchen Light On |
| H | Living room Curtains Close | Bedroom Light Off |
| I | Bedroom Curtains Open | Living room Light brightness = 60% |
| J | Outdoor temperature > 27°C | Bedroom Door Open |
| K | Living room temperature > 25°C | Bedroom Window Open |
| L | Living room AC On | Bedroom AC Off |

Action Reversal Rule Execution Order

| Set 1 (ARB) | Set 2 (ARF) |
|---|---|
| Rule 2 | Rule 1 |
| Rule 5 | Rule 4 |
| Rule 3 | Rule 2 |
| Rule 6 | Rule 5 |
| Rule 1 | Rule 3 |
| Rule 4 | Rule 6 |
| Rule 5 | Rule 4 |

**Figure A.2:** Rule sets for *Action Reversal* bug. Set 1 is used for the *Baseline* version and Set 2 for *Filter* version of the dashboard. Conflicting rules are highlighted.

Condition Unsatisfiable TAP Rules

| Rule Description | Set 1 (CUB) | Set 2 (CUF) |
|---|---|---|
| If (A) then do (NOT B) | Rule 5 | Rule 4 |
| If (B) then do (NOT A) | Rule 1 | Rule 2 |
| If (C), while (A AND D AND B) holds then do (E) | Rule 4 | Rule 5 |
| If (F), while (G) holds then do (H) | Rule 2 | Rule 3 |
| If (G), while (F) holds then do (H) | Rule 6 | Rule 6 |
| If (I) then do (J) | Rule 3 | Rule 1 |

Condition Unsatisfiable Variables

| Variables | Set 1 (CUB) | Set 2 (CUF) |
|---|---|---|
| A | Living room Curtains Open | Bedroom AC On |
| B | Living room Light On | Bedroom Window Open |
| C | Outdoor brightness < 20% | Outdoor thermostat temperature >= 28°C |
| D | Living room Light brightness < 50% | Bedroom AC temperature > 22°C |
| E | Living room Light brightness = 80% | Bedroom AC temperature = 22°C |
| F | Outdoor temperature > 30°C | Outdoor brightness < 40% |
| G | Living room temperature > 26°C | Bedroom brightness < 50% |
| H | Living Room AC On | Bedroom Light On |
| I | Bedroom AC on | Living room Curtains Open |
| J | Bedroom Door Close | Living room Light Off |

Condition Unsatisfiable Rule Execution Order

| Set 1 (CUB) | Set 2 (CUF) |
|---|---|
| Rule 2 | Rule 3 |
| Rule 1 | Rule 2 |
| Rule 3 | Rule 1 |
| Rule 5 | Rule 4 |
| Rule 4 | Rule 5 |
| Rule 6 | Rule 6 |

**Figure A.3:** Rule sets for *Condition Unsatisfiable* bug. Set 1 is used for the *Baseline* version and Set 2 for *Filter* version of the dashboard. Conflicting rules are highlighted.

Infinite Loop TAP Rules

| Rule Description | Set 1 (ILB) | Set 2 (ILF) |
|---|---|---|
| If (A) then do (B) | Rule 4 | Rule 6 |
| If (B), while (E) holds then do (C) | Rule 8 | Rule 3 |
| If (C), while (F) holds then do (NOT A) | Rule 1 | Rule 7 |
| If (NOT A) then do (D) | Rule 5 | Rule 4 |
| If (D), while (H) holds then do (E) | Rule 2 | Rule 8 |
| If (E), while (G) holds then do (A) | Rule 6 | Rule 1 |
| If (X) then do (Y) | Rule 3 | Rule 5 |
| If (F) then do (Z) | Rule 7 | Rule 2 |

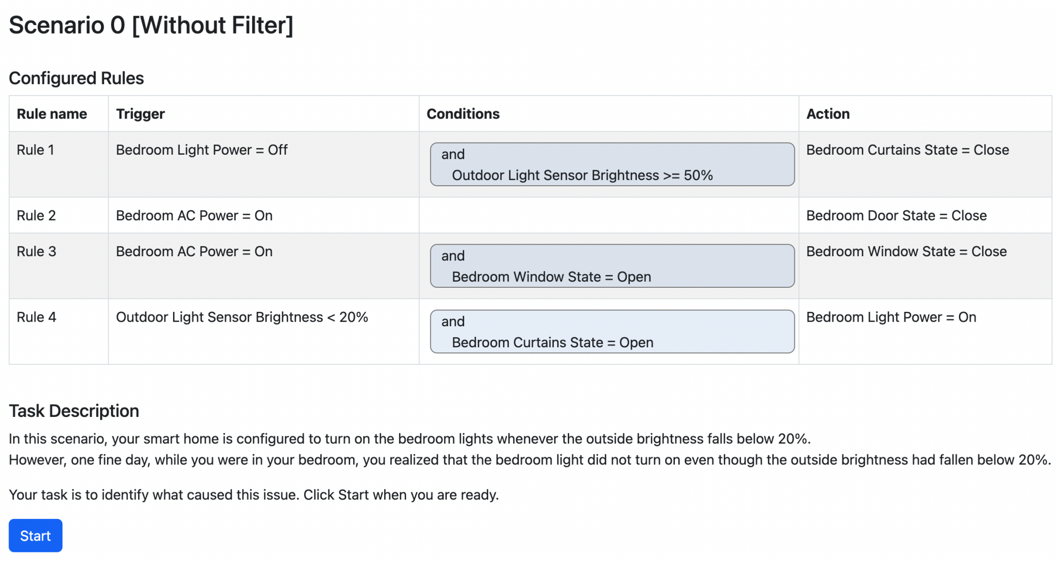
Infinite Loop Variables

| Description | Set 1 (ILB) | Set 2 (ILF) |
|---|---|---|
| A | Bedroom Light On | Living room AC On |
| B | Bedroom AC On | Living room Window Close |
| C | Bedroom Door Close | Living room Light Off |
| D | Bedroom Curtains Open | Living room Curtains Close |
| E | Bedroom Window Close | Living Room TV On |
| F | Outdoor brightness > 60% | Outdoor temperature < 25°C |
| G | Bedroom Light sensor brightness < 50% | Living room thermostat temperature > 26°C |
| H | Outdoor Thermostat temperature < 20°C | Living room brightness < 30% |
| X | Kitchen Thermostat temperature > 28°C | Bedroom Light On |
| Y | Kitchen Window Open | Bedroom Window Close |
| Z | Living room Curtains Open | Bedroom AC Off |

Infinite Loop Rule Execution Order

| Set 1 (ILB) | Set 2 (ILF) |
|---|---|
| Rule 7 | Rule 2 |
| Rule 3 | Rule 5 |
| Rule 7 | Rule 2 |
| Rule 6 | Rule 1 |
| Rule 4 | Rule 6 |
| Rule 8 | Rule 3 |
| Rule 1 | Rule 7 |
| Rule 5 | Rule 4 |
| Rule 2 | Rule 8 |
| Loop 6, 4, 8, 1, 5, 2, 6, … | Loop 1, 6, 3, 7, 4, 8, 1, … |

**Figure A.4:** Rule sets for *Infinite Loop* bug. Set 1 is used for the *Baseline* version and Set 2 for *Filter* version of the dashboard. Conflicting rules are highlighted.



Scenario 0 [Without Filter]

Configured Rules

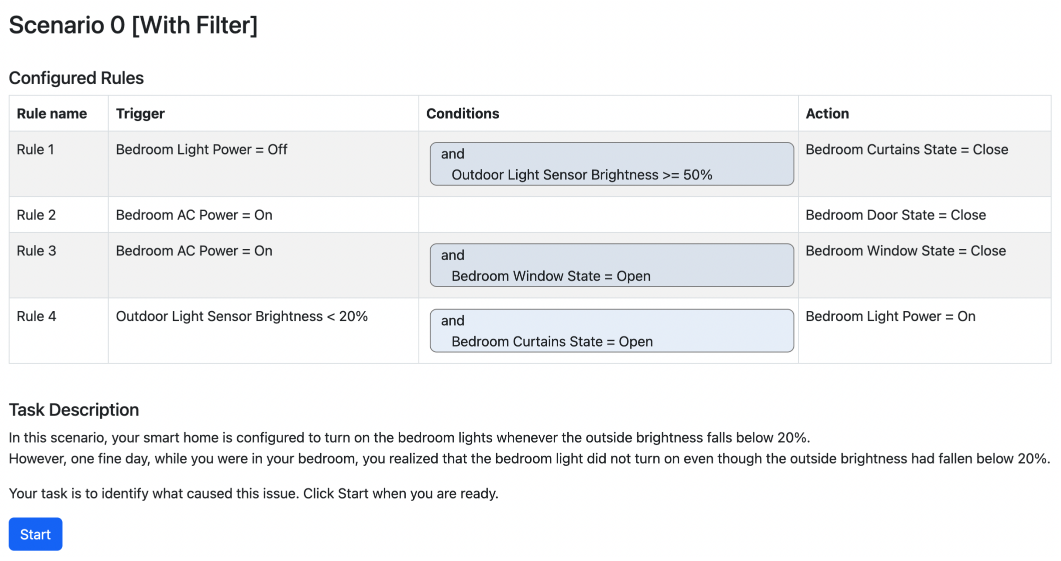
| Rule name | Trigger | Conditions | Action |
|---|---|---|---|
| Rule 1 | Bedroom Light Power = Off | and Outdoor Light Sensor Brightness >= 50% | Bedroom Curtains State = Close |
| Rule 2 | Bedroom AC Power = On | | Bedroom Door State = Close |
| Rule 3 | Bedroom AC Power = On | and Bedroom Window State = Open | Bedroom Window State = Close |
| Rule 4 | Outdoor Light Sensor Brightness < 20% | and Bedroom Curtains State = Open | Bedroom Light Power = On |

Task Description

In this scenario, your smart home is configured to turn on the bedroom lights whenever the outside brightness falls below 20%.
However, one fine day, while you were in your bedroom, you realized that the bedroom light did not turn on even though the outside brightness had fallen below 20%.

Your task is to identify what caused this issue. Click Start when you are ready.

**Figure A.5:** Study Instruction Page for *Baseline* dashboard task.



Scenario 0 [With Filter]

Configured Rules

| Rule name | Trigger | Conditions | Action |
|---|---|---|---|
| Rule 1 | Bedroom Light Power = Off | and Outdoor Light Sensor Brightness >= 50% | Bedroom Curtains State = Close |
| Rule 2 | Bedroom AC Power = On | | Bedroom Door State = Close |
| Rule 3 | Bedroom AC Power = On | and Bedroom Window State = Open | Bedroom Window State = Close |
| Rule 4 | Outdoor Light Sensor Brightness < 20% | and Bedroom Curtains State = Open | Bedroom Light Power = On |

Task Description

In this scenario, your smart home is configured to turn on the bedroom lights whenever the outside brightness falls below 20%.
However, one fine day, while you were in your bedroom, you realized that the bedroom light did not turn on even though the outside brightness had fallen below 20%.

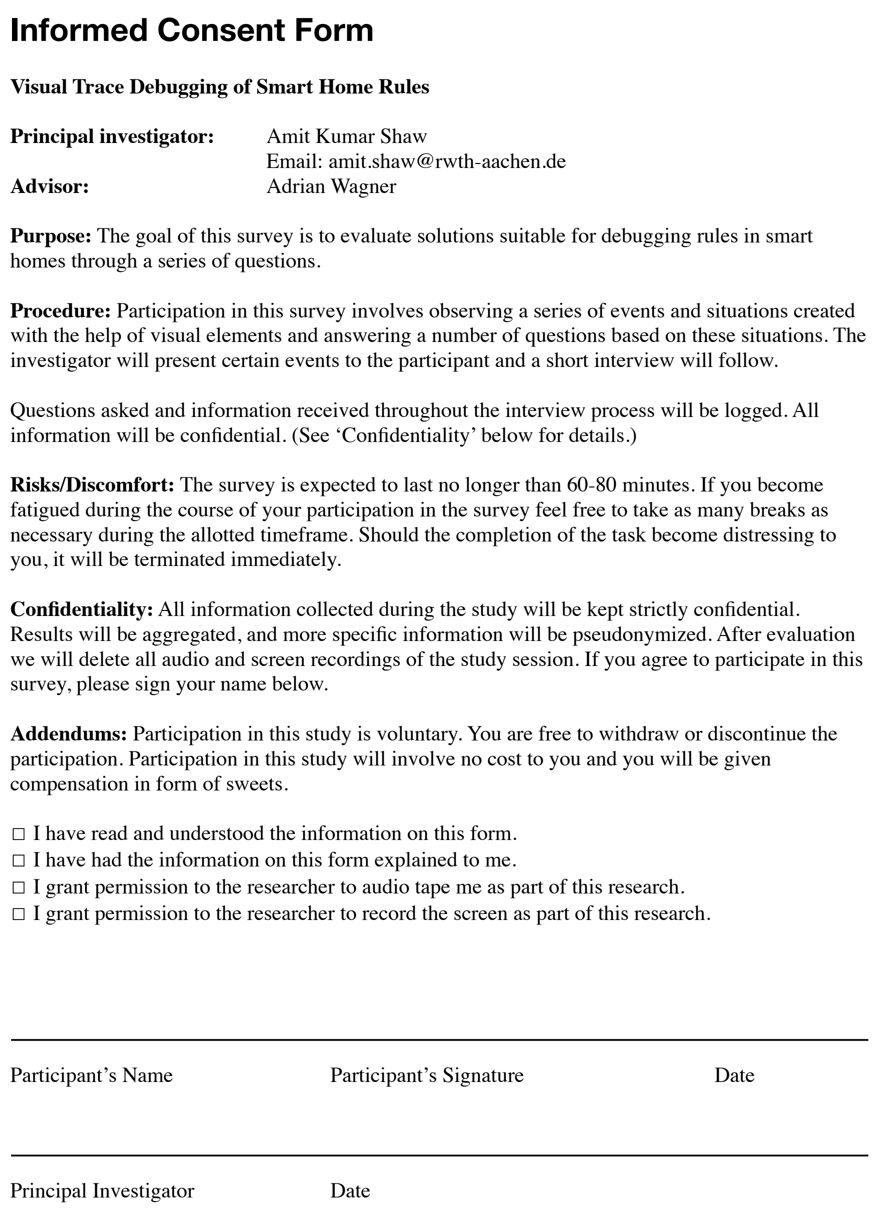
Your task is to identify what caused this issue. Click Start when you are ready.

**Figure A.6:** Study Instruction Page for *Filter* dashboard task.

# Appendix B

# User Study Documents

We present the documents used during the user study. Figure B.1 shows the informed consent form signed by each participant before the start of the user study. Figures B.2 and B.3 show the demographics survey form completed by each participant before the user study. Figure B.4 shows the feedback form filled out by the participants after completing each task. Figures B.5 and B.6 show the System Causability Scale questions for *Baseline* and *Filter* dashboards, respectively. Figure B.7 shows the feedback form filled out by the participants at the end of the user study.



## Informed Consent Form

**Visual Trace Debugging of Smart Home Rules**

| | |
|---|---|
| **Principal investigator:** | Amit Kumar Shaw<br>Email: amit.shaw@rwth-aachen.de |
| **Advisor:** | Adrian Wagner |

**Purpose:** The goal of this survey is to evaluate solutions suitable for debugging rules in smart homes through a series of questions.

**Procedure:** Participation in this survey involves observing a series of events and situations created with the help of visual elements and answering a number of questions based on these situations. The investigator will present certain events to the participant and a short interview will follow.

Questions asked and information received throughout the interview process will be logged. All information will be confidential. (See 'Confidentiality' below for details.)

**Risks/Discomfort:** The survey is expected to last no longer than 60-80 minutes. If you become fatigued during the course of your participation in the survey feel free to take as many breaks as necessary during the allotted timeframe. Should the completion of the task become distressing to you, it will be terminated immediately.

**Confidentiality:** All information collected during the study will be kept strictly confidential. Results will be aggregated, and more specific information will be pseudonymized. After evaluation we will delete all audio and screen recordings of the study session. If you agree to participate in this survey, please sign your name below.

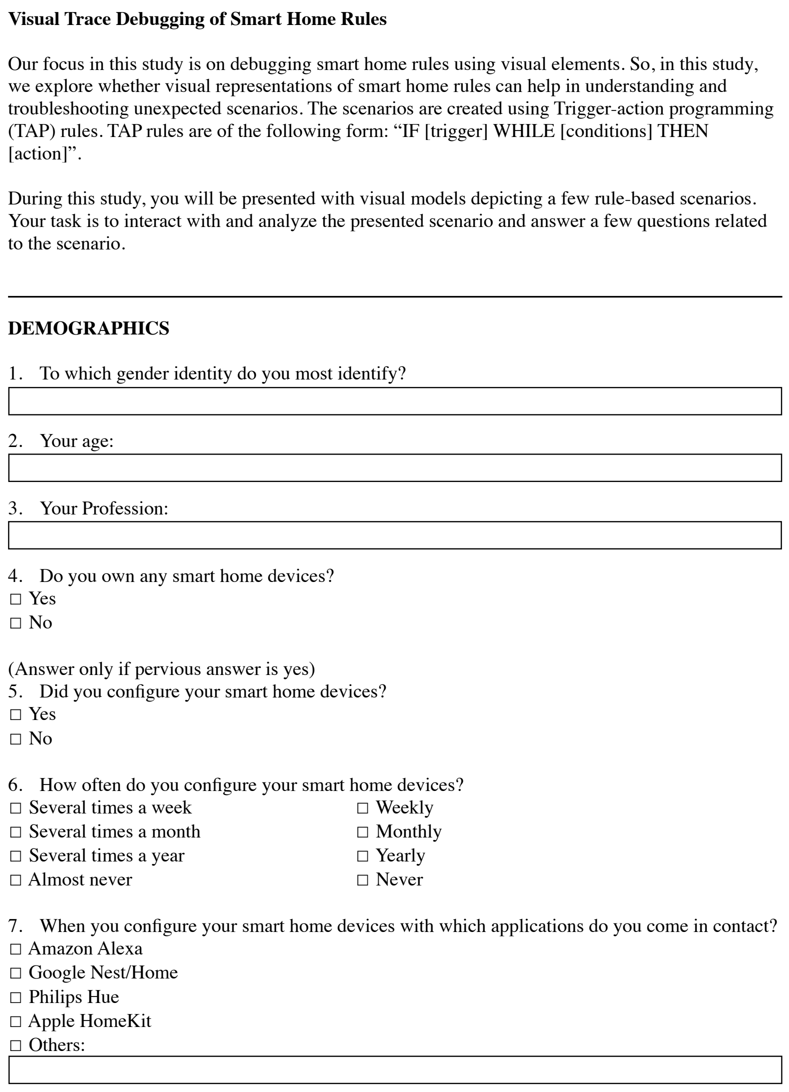
**Addendums:** Participation in this study is voluntary. You are free to withdraw or discontinue the participation. Participation in this study will involve no cost to you and you will be given compensation in form of sweets.

□ I have read and understood the information on this form.
□ I have had the information on this form explained to me.
□ I grant permission to the researcher to audio tape me as part of this research.
□ I grant permission to the researcher to record the screen as part of this research.

Participant's Name    Participant's Signature    Date

Principal Investigator    Date

**Figure B.1:** Informed Consent Form.



**Visual Trace Debugging of Smart Home Rules**

Our focus in this study is on debugging smart home rules using visual elements. So, in this study, we explore whether visual representations of smart home rules can help in understanding and troubleshooting unexpected scenarios. The scenarios are created using Trigger-action programming (TAP) rules. TAP rules are of the following form: "IF [trigger] WHILE [conditions] THEN [action]".

During this study, you will be presented with visual models depicting a few rule-based scenarios. Your task is to interact with and analyze the presented scenario and answer a few questions related to the scenario.

---

**DEMOGRAPHICS**

1. To which gender identity do you most identify?

2. Your age:

3. Your Profession:

4. Do you own any smart home devices?
☐ Yes
☐ No

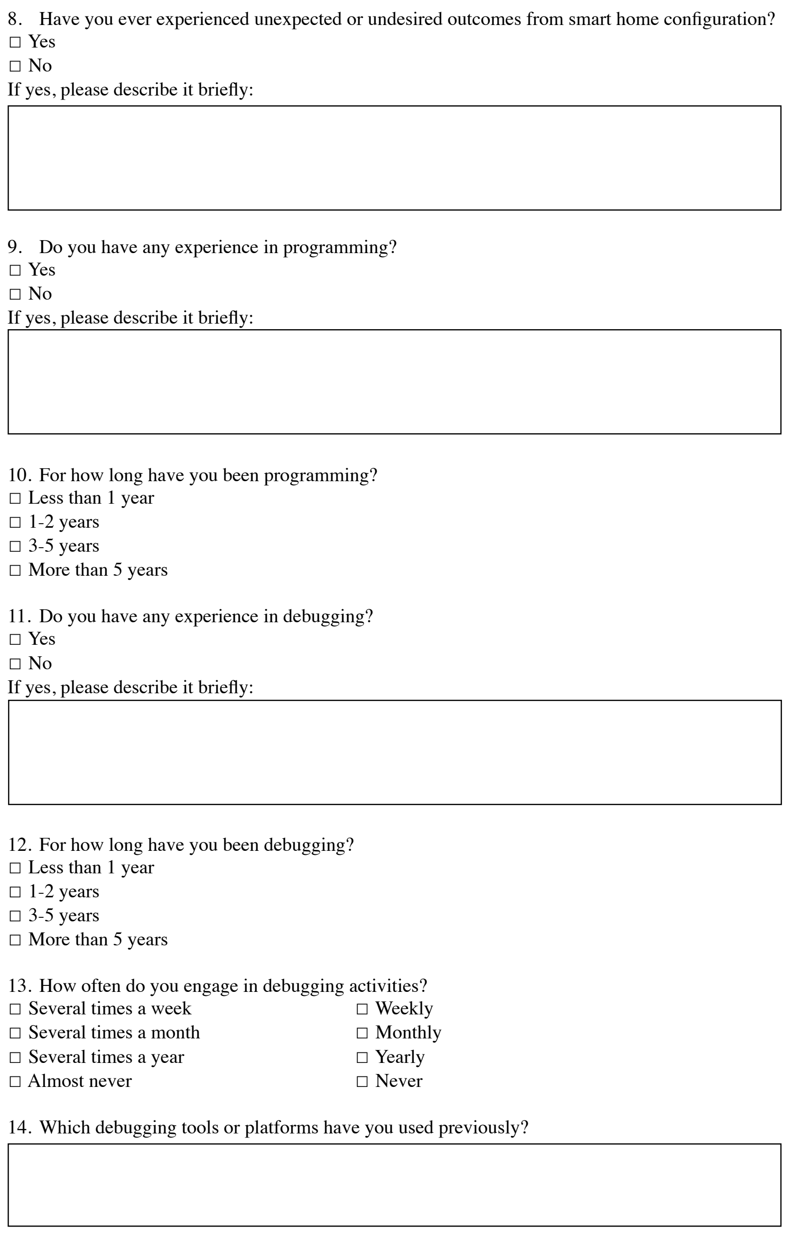
(Answer only if pervious answer is yes)
5. Did you configure your smart home devices?
☐ Yes
☐ No

6. How often do you configure your smart home devices?

| | |
|---|---|
| ☐ Several times a week | ☐ Weekly |
| ☐ Several times a month | ☐ Monthly |
| ☐ Several times a year | ☐ Yearly |
| ☐ Almost never | ☐ Never |

7. When you configure your smart home devices with which applications do you come in contact?
☐ Amazon Alexa
☐ Google Nest/Home
☐ Philips Hue
☐ Apple HomeKit
☐ Others:

**Figure B.2:** Demographics survey form (1).



8. Have you ever experienced unexpected or undesired outcomes from smart home configuration?
□ Yes
□ No
If yes, please describe it briefly:

9. Do you have any experience in programming?
□ Yes
□ No
If yes, please describe it briefly:

10. For how long have you been programming?
□ Less than 1 year
□ 1-2 years
□ 3-5 years
□ More than 5 years

11. Do you have any experience in debugging?
□ Yes
□ No
If yes, please describe it briefly:

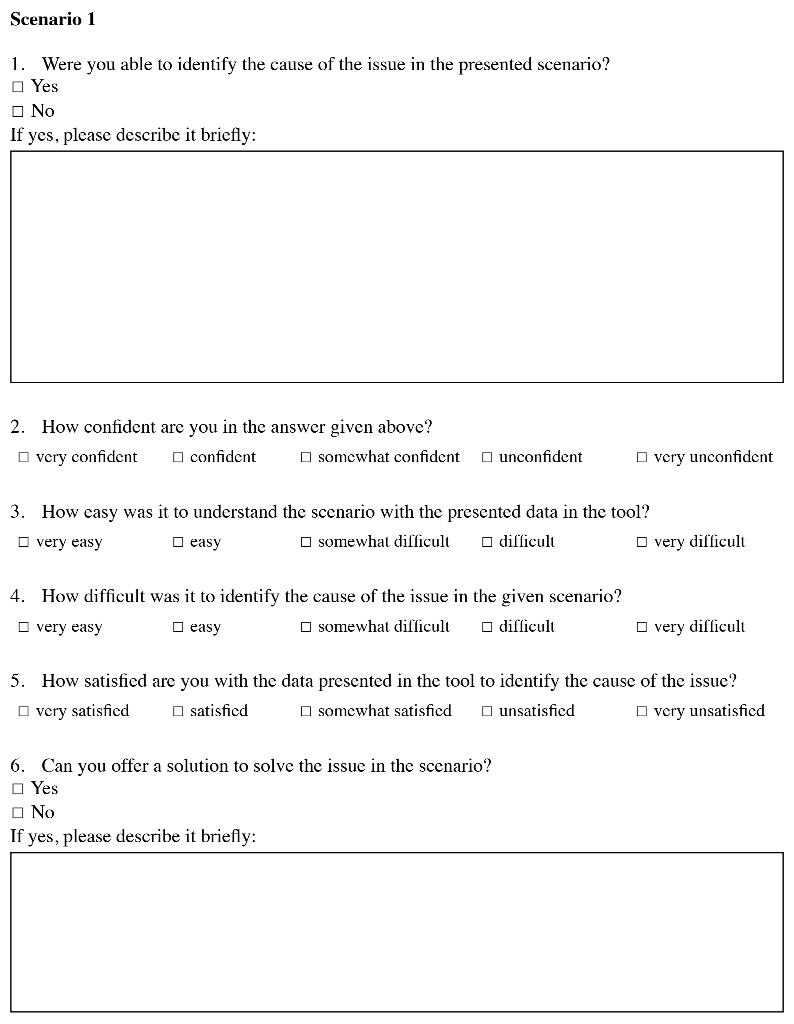
12. For how long have you been debugging?
□ Less than 1 year
□ 1-2 years
□ 3-5 years
□ More than 5 years

13. How often do you engage in debugging activities?

| | |
|---|---|
| □ Several times a week | □ Weekly |
| □ Several times a month | □ Monthly |
| □ Several times a year | □ Yearly |
| □ Almost never | □ Never |

14. Which debugging tools or platforms have you used previously?

**Figure B.3:** Demographics survey form (2).



**Scenario 1**

1. Were you able to identify the cause of the issue in the presented scenario?

☐ Yes
☐ No

If yes, please describe it briefly:

2. How confident are you in the answer given above?

| | | | | |
|---|---|---|---|---|
| ☐ very confident | ☐ confident | ☐ somewhat confident | ☐ unconfident | ☐ very unconfident |

3. How easy was it to understand the scenario with the presented data in the tool?

| | | | | |
|---|---|---|---|---|
| ☐ very easy | ☐ easy | ☐ somewhat difficult | ☐ difficult | ☐ very difficult |

4. How difficult was it to identify the cause of the issue in the given scenario?

| | | | | |
|---|---|---|---|---|
| ☐ very easy | ☐ easy | ☐ somewhat difficult | ☐ difficult | ☐ very difficult |

5. How satisfied are you with the data presented in the tool to identify the cause of the issue?

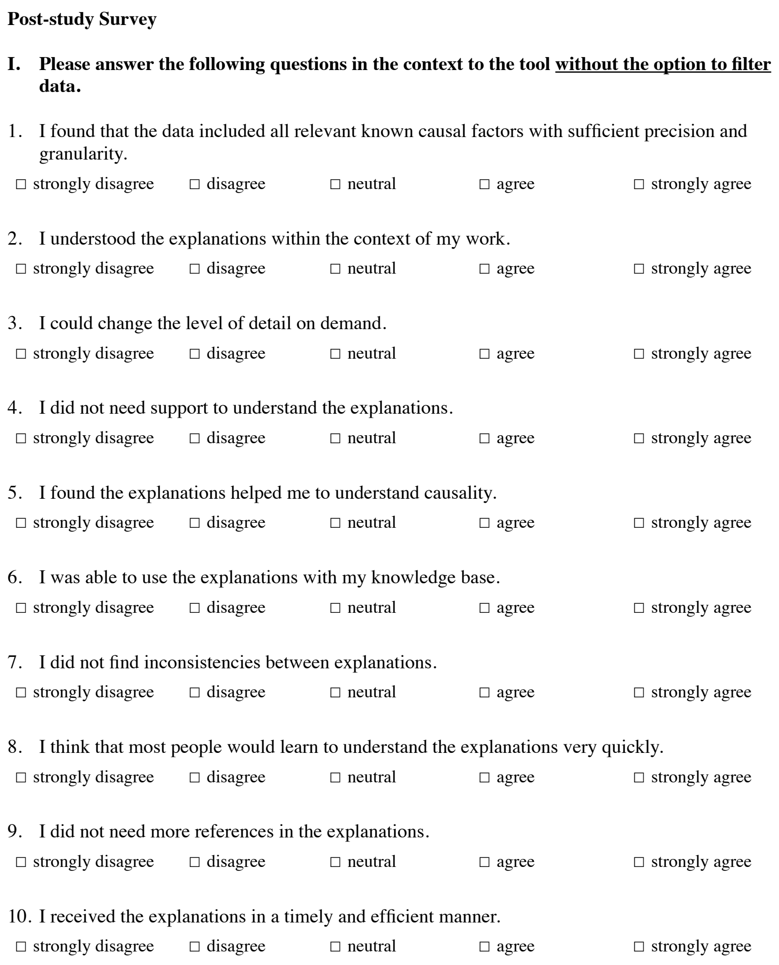
| | | | | |
|---|---|---|---|---|
| ☐ very satisfied | ☐ satisfied | ☐ somewhat satisfied | ☐ unsatisfied | ☐ very unsatisfied |

6. Can you offer a solution to solve the issue in the scenario?

☐ Yes
☐ No

If yes, please describe it briefly:

**Figure B.4:** Form to record feedback after each task.



**Post-study Survey**

**I. Please answer the following questions in the context to the tool <u>without the option to filter</u> data.**

1. I found that the data included all relevant known causal factors with sufficient precision and granularity.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

2. I understood the explanations within the context of my work.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

3. I could change the level of detail on demand.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

4. I did not need support to understand the explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

5. I found the explanations helped me to understand causality.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

6. I was able to use the explanations with my knowledge base.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

7. I did not find inconsistencies between explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

8. I think that most people would learn to understand the explanations very quickly.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

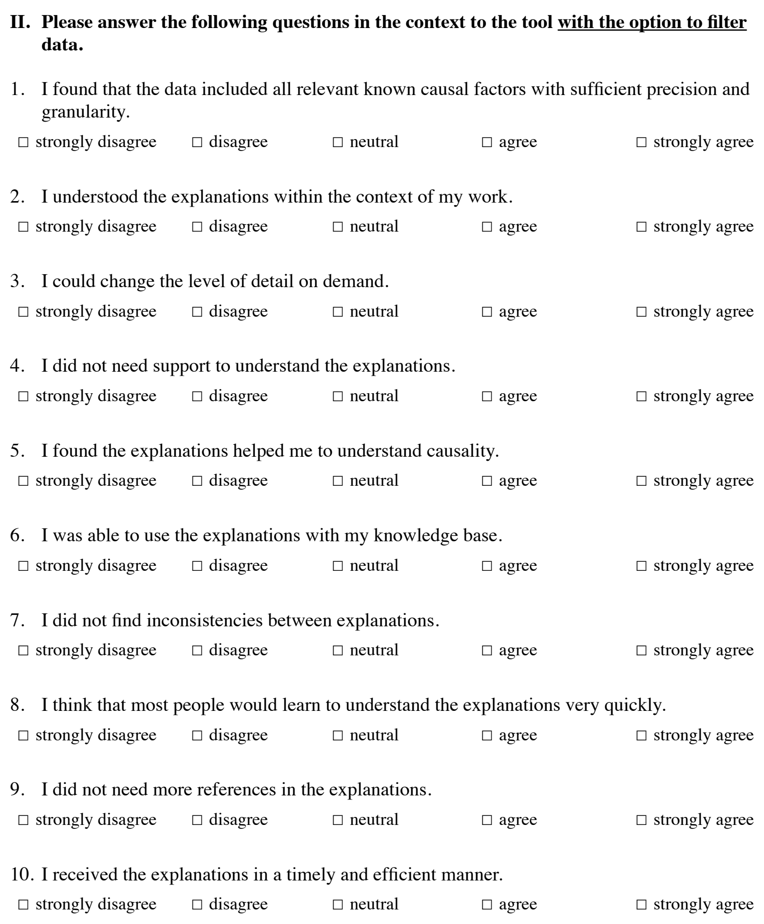
9. I did not need more references in the explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

10. I received the explanations in a timely and efficient manner.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

**Figure B.5:** System Causability Scale questions for *Baseline* dashboard.



**II. Please answer the following questions in the context to the tool <u>with the option to filter</u> data.**

1. I found that the data included all relevant known causal factors with sufficient precision and granularity.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

2. I understood the explanations within the context of my work.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

3. I could change the level of detail on demand.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

4. I did not need support to understand the explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

5. I found the explanations helped me to understand causality.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

6. I was able to use the explanations with my knowledge base.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

7. I did not find inconsistencies between explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

8. I think that most people would learn to understand the explanations very quickly.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

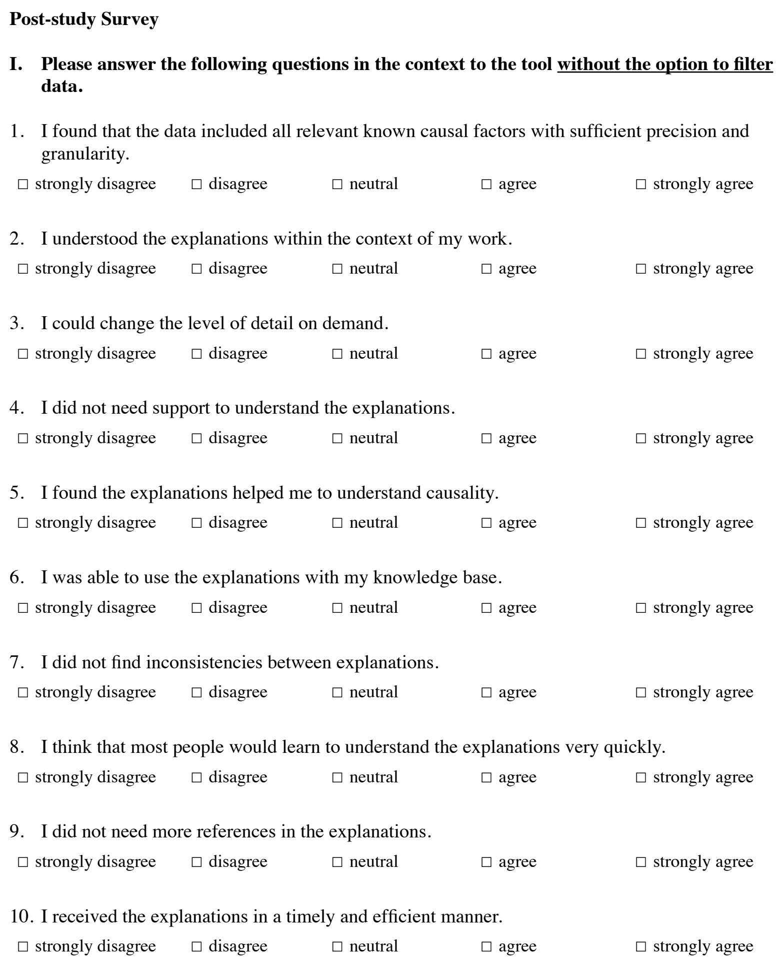
9. I did not need more references in the explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

10. I received the explanations in a timely and efficient manner.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

**Figure B.6:** System Causability Scale questions for *Filter* dashboard.



**Post-study Survey**

**I. Please answer the following questions in the context to the tool <u>without the option to filter</u> data.**

1. I found that the data included all relevant known causal factors with sufficient precision and granularity.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

2. I understood the explanations within the context of my work.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

3. I could change the level of detail on demand.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

4. I did not need support to understand the explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

5. I found the explanations helped me to understand causality.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

6. I was able to use the explanations with my knowledge base.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

7. I did not find inconsistencies between explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

8. I think that most people would learn to understand the explanations very quickly.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

9. I did not need more references in the explanations.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

10. I received the explanations in a timely and efficient manner.

☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree

**Figure B.7:** Post Study Questionnaire.

# Bibliography

Abdullah Al Farooq, Ehab Al-Shaer, Thomas Moyer, and Krishna Kant. Iotc2: A formal method approach for detecting conflicts in large scale iot systems. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, page 442–447, April 2019. URL `https://ieeexplore.ieee.org/abstract/document/8717844`.

Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. How users interpret bugs in trigger-action programming. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–12, New York, NY, USA, May 2019. Association for Computing Machinery. ISBN 978-1-4503-5970-2. doi: 10.1145/3290605.3300782. URL `https://dl.acm.org/doi/10.1145/3290605.3300782`.

John Brooke. Sus: a "quick and dirty'usability. *Usability evaluation in industry*, 189(3):189–194, 1996.

A.J. Bernheim Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. Home automation in the wild: challenges and opportunities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, page 2115–2124, New York, NY, USA, May 2011. Association for Computing Machinery. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979249. URL `https://dl.acm.org/doi/10.1145/1978942.1979249`.

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): 124:1–124:24, December 2020. doi: 10.1145/3432225.

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):154:1–154:20, June 2022. doi: 10.1145/3532204.

Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. Empowering end users in debugging trigger-action rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–13, New York, NY, USA, May 2019. Association for Computing Machinery. ISBN 978-1-4503-5970-2. doi: 10.1145/3290605.3300618. URL `https://dl.acm.org/doi/10.1145/3290605.3300618`.

Luigi De Russis and Fulvio Corno. Homerules: A tangible end-user programming interface for smart homes. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, page 2109–2114, New York, NY, USA, April 2015. Association for Computing Machinery. ISBN 978-1-4503-3146-3. doi: 10.1145/2702613.2732795. URL `https://dl.acm.org/doi/10.1145/2702613.2732795`.

Luigi De Russis and Alberto Monge Roffarello. A debugging approach for trigger-action programming. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI EA '18, page 1–6, New York, NY, USA, April 2018. Association for Computing Machinery. ISBN 978-1-4503-5621-3. doi: 10.1145/3170427.3188641. URL `https://dl.acm.org/doi/10.1145/3170427.3188641`.

Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. Personalization of context-dependent applications through trigger-action rules. *ACM Transactions on Computer-Human Interaction*, 24(2):14:1–14:33, April 2017. ISSN 1073-0516. doi: 10.1145/3057861.

Andreas Holzinger, André Carrington, and Heimo Müller. Measuring the quality of explanations: The system causability scale (scs). *KI - Künstliche Intelligenz*, 34(2):

193–198, June 2020. ISSN 1610-1987. doi: 10.1007/s13218-020-00636-z.

Karl A. Hribernik, Zied Ghrairi, Carl Hans, and Klaus-Dieter Thoben. Co-creating the internet of things — first experiences in the participatory design of intelligent products with arduino. page 1–9, June 2011. URL `https://ieeexplore.ieee.org/abstract/document/6041235`.

Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, page 215–225, New York, NY, USA, September 2015. Association for Computing Machinery. ISBN 978-1-4503-3574-4. doi: 10.1145/2750858.2805830. URL `https://dl.acm.org/doi/10.1145/2750858.2805830`.

Amy J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, page 151–158, New York, NY, USA, April 2004. Association for Computing Machinery. ISBN 978-1-58113-702-6. doi: 10.1145/985692.985712. URL `https://dl.acm.org/doi/10.1145/985692.985712`.

Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-User Development: An Emerging Paradigm*, page 1–8. Human-Computer Interaction Series. Springer Netherlands, Dordrecht, 2006. ISBN 978-1-4020-5386-3. doi: 10.1007/1-4020-5386-X_1. URL `https://doi.org/10.1007/1-4020-5386-X_1`.

Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

Meiyi Ma, Sarah Masud Preum, and John A. Stankovic. Cityguard: A watchdog for safety-aware conflict detection in smart cities. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, IoTDI '17, page 259–270, New York, NY, USA, April 2017. Association for Computing Machinery. ISBN 978-1-4503-4966-6. doi: 10.1145/3054977.

3054989. URL `https://dl.acm.org/doi/10.1145/3054977.3054989`.

Evan Magill and Jesse Blum. Exploring conflicts in rule-based sensor networks. *Pervasive and Mobile Computing*, 27:133–154, April 2016. ISSN 1574-1192. doi: 10.1016/j.pmcj.2015.08.005.

Davit Marikyan, Savvas Papagiannidis, and Eleftherios Alamanos. A systematic review of the smart home literature: A user perspective. *Technological Forecasting and Social Change*, 138:139–154, January 2019. ISSN 0040-1625. doi: 10.1016/j.techfore.2018.08.015.

Sarah Mennicken and Elaine M. Huang. *Hacking the Natural Habitat: An In-the-Wild Study of Smart Homes, Their Development, and the People Who Live in Them*, volume 7319 of *Lecture Notes in Computer Science*, page 143–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31204-5. doi: 10.1007/978-3-642-31205-2_10. URL `http://link.springer.com/10.1007/978-3-642-31205-2_10`.

Alessandro A. Nacci, Vincenzo Rana, Bharathan Balaji, Paola Spoletini, Rajesh Gupta, Donatella Sciuto, and Yuvraj Agarwal. Buildingrules: A trigger-action–based system to manage complex commercial buildings. *ACM Transactions on Cyber-Physical Systems*, 2(2):13:1–13:22, May 2018. ISSN 2378-962X. doi: 10.1145/3185500.

Shahrokh Nikou. Factors driving the adoption of smart home technology: An empirical assessment. *Telematics and Informatics*, 45:101283, December 2019. ISSN 0736-5853. doi: 10.1016/j.tele.2019.101283.

Steven P. Reiss. Trace-based debugging. In Peter A. Fritzson, editor, *Automated and Algorithmic Debugging*, Lecture Notes in Computer Science, page 305–314, Berlin, Heidelberg, 1993. Springer. ISBN 978-3-540-48141-6. doi: 10.1007/BFb0019416.

Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI

’14, page 803–812, New York, NY, USA, April 2014. Association for Computing Machinery. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557420. URL `https://dl.acm.org/doi/10.1145/2556288.2557420`.

Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI ’16, page 3227–3231, New York, NY, USA, May 2016. Association for Computing Machinery. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858556. URL `https://dl.acm.org/doi/10.1145/2858036.2858556`.

Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’19, page 1439–1453, New York, NY, USA, November 2019. Association for Computing Machinery. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3345662. URL `https://dl.acm.org/doi/10.1145/3319535.3345662`.

Jong-bum Woo and Youn-kyung Lim. User experience in do-it-yourself-style smart homes. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp ’15, page 779–790, New York, NY, USA, September 2015. Association for Computing Machinery. ISBN 978-1-4503-3574-4. doi: 10.1145/2750858.2806063. URL `https://dl.acm.org/doi/10.1145/2750858.2806063`.

Imam Nur Bani Yusuf, Lingxiao Jiang, and David Lo. Accurate generation of trigger-action programs with domain-adapted sequence-to-sequence learning. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC ’22, page 99–110, New York, NY, USA, October 2022. Association for Computing Machinery. ISBN 978-1-4503-9298-3. doi: 10.1145/3524610.3527922. URL `https://dl.acm.org/doi/10.1145/3524610.3527922`.

Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. Autotap: Synthesizing and repairing trigger-action programs using ltl properties. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, page 281–291, May 2019. doi: 10.1109/ICSE.2019.00043.

Lefan Zhang, Weijia He, Olivia Morkved, Valerie Zhao, Michael L. Littman, Shan Lu, and Blase Ur. Trace2tap: Synthesizing trigger-action programs from traces of behavior. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3):104:1–104:26, September 2020. doi: 10.1145/3411838.

Lefan Zhang, Cyrus Zhou, Michael L. Littman, Blase Ur, and Shan Lu. Helping users debug trigger-action programs. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(4):196:1–196:32, January 2023. doi: 10.1145/3569506.

Valerie Zhao, Lefan Zhang, Bo Wang, Shan Lu, and Blase Ur. Visualizing differences to improve end-user understanding of trigger-action programs. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI EA '20, page 1–10, New York, NY, USA, April 2020. Association for Computing Machinery. ISBN 978-1-4503-6819-3. doi: 10.1145/3334480.3382940. URL `https://dl.acm.org/doi/10.1145/3334480.3382940`.

Valerie Zhao, Lefan Zhang, Bo Wang, Michael L. Littman, Shan Lu, and Blase Ur. Understanding trigger-action programs through novel visualizations of program differences. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, page 1–17, New York, NY, USA, May 2021. Association for Computing Machinery. ISBN 978-1-4503-8096-6. doi: 10.1145/3411764.3445567. URL `https://dl.acm.org/doi/10.1145/3411764.3445567`.

# Index

Typeset February 2, 2024