Helping Users Debug Trigger-Action Programs

LEFAN ZHANG, University of Chicago, USA
CYRUS ZHOU, University of Chicago, USA
MICHAEL L. LITTMAN, Brown University, USA
BLASE UR, University of Chicago, USA
SHAN LU, University of Chicago, USA

Trigger-action programming (TAP) empowers a wide array of users to automate Internet of Things (IoT) devices. However, it can be challenging for users to create completely correct trigger-action programs (TAPs) on the first try, necessitating debugging. While TAP has received substantial research attention, TAP debugging has not. In this paper, we present the first empirical study of users' end-to-end TAP debugging process, focusing on obstacles users face in debugging TAPs and how well users ultimately fix incorrect automations. To enable this study, we added TAP capabilities to an existing 3-D smart home simulator. Thirty remote participants spent a total of 84 hours debugging TAPs using this simulator. Without additional support, participants were often unable to fix buggy TAPs due to a series of obstacles we document. However, we also found that two novel tools we developed helped participants overcome many of these obstacles and more successfully debug TAPs. These tools collect either implicit or explicit feedback from users about automations that should or should not have happened in the past, using a SAT-solving-based algorithm we developed to automatically modify the TAPs to account for this feedback.

CCS Concepts: • **Human-centered computing** → **Ubiquitous and mobile computing**; **Empirical studies in HCI**; *Empirical studies in interaction design*; • **Software and its engineering** → *Designing software*; General programming languages; Virtual worlds software; *Software evolution.*

Additional Key Words and Phrases: Trigger-action programming, end-user programming, end-user debugging, smart environment, symbolic reasoning, IoT, Internet of Things, IFTTT

ACM Reference Format:
Lefan Zhang, Cyrus Zhou, Michael L. Littman, Blase Ur, and Shan Lu. 2022. Helping Users Debug Trigger-Action Programs. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 6, 4, Article 196 (December 2022), 32 pages. https://doi.org/10.1145/3569506

1 INTRODUCTION

Internet of Things (**IoT**) smart devices have become common in homes [22]. Through end-user programming tools, even users without programming experience can personalize and automate their devices. A popular approach to end-user programming is trigger-action programming (**TAP**), which is supported by IFTTT [24], Mozilla's Things Gateway [20], Samsung SmartThings [28], Microsoft Flow [24], OpenHab [30], Home Assistant [17], Ripple [7], Zapier [24], and others. With TAP, in a graphical interface users create event-driven rules (termed **TAPs**, short for trigger-action programs) following the form "IF *trigger* occurs, THEN perform *action*." An example rule might be "IF it starts raining, THEN turn the lights blue."

Authors' addresses: Lefan Zhang, lefanz@uchicago.edu, University of Chicago, Chicago, Illinois, USA, 60637; Cyrus Zhou, zhouzk@uchicago.edu, University of Chicago, Chicago, Illinois, USA, 60637; Michael L. Littman, mlittman@cs.brown.edu, Brown University, Providence, Rhode Island, USA, 02912; Blase Ur, blase@uchicago.edu, University of Chicago, Chicago, Illinois, USA, 60637; Shan Lu, shanlu@uchicago.edu, University of Chicago, Chicago, Illinois, USA, 60637.

2474-9567/2022/12-ART196
https://doi.org/10.1145/3569506

Although TAP excels in solving simple automation tasks [35], prior work has shown that users suffer from misunderstandings and bugs when using TAP in complex scenarios [3, 19]. Furthermore, in real-world studies, some participants struggled to write correct TAP rules even for common daily tasks [41]. Thus, it can be hard for users to develop correct TAPs with a single attempt in complex and nuanced automation scenarios.

As a result, TAP *debugging* is thus a process that users will frequently encounter when deploying TAPs in real situations. Despite a rich and growing literature on TAP, to our knowledge no prior work has studied the end-to-end process of debugging TAPs. That is, while prior work has studied users' misunderstandings about TAP in highly controlled scenarios [3, 19] or designed algorithms and tools for identifying possible bugs [4, 10, 25, 26, 29, 37, 43], these prior studies did not examine how users move from experiencing incorrect automation behaviors to trying to pinpoint the issue to trying to fix problematic TAPs. It remains unclear what exact obstacles users may face at each stage of debugging.

While not understanding the full end-to-end TAP debugging process is a key gap in the literature, the existence of this gap is much less surprising when considering the substantial hurdles to conducting rigorous studies on this topic. The logistics of a real-world TAP debugging study are daunting, from the cost of purchasing smart devices to the time required to temporarily deploy these devices in participants' homes to the justifiable privacy concerns of potential participants. Furthermore, for each participant, TAPs might only be triggered a couple of times a day, so many days or weeks of interaction may be required before even a single debugging scenario emerges. Making things worse, it is extremely hard to compare different TAP debugging tools in a controlled way because participants have different home layouts, device usage patterns, TAPs, and expectations.

Based on debugging approaches in contexts other than end-user programming, we hypothesized that TAP debugging would encompass the following four-step process, which the "Stages" column in Figure 1 also illustrates:

(I) The user experiences and identifies an unexpected or incorrect behavior of their automated device(s);
(II) The user examines the TAPs to localize the fault, identifying a possible cause of the unexpected behavior(s);
(III) The user proposes a modification (which we term a **patch**) to the TAPs aiming to resolve the problem;
(IV) The user attempts to refine the initial patch to ensure it matches the intended behavior.

Prior work has only examined parts of this end-to-end debugging process in isolation. Previous papers have provided insights into automated bug detection [4, 10, 25, 26, 29, 37, 41] (related to Stage II), automated TAP synthesis [41, 42] (related to Stage III), and TAP visualization [10, 26, 43, 44]. These existing tools are not designed to help users through the whole process of fixing TAPs based on misbehavior(s) they experience.

Contributions

To better understand and support end-to-end TAP debugging, this paper makes the following contributions:
1) Design and implementation of new TAP debugging interfaces and corresponding algorithms. To improve user support at every stage of TAP debugging (Figure 1), we design and implement the following novel user interface components, those interfaces' underlying algorithms, and additional analysis tools:

- A **History Visualization** interface (Figure 2) that allows users to efficiently review automation events and contexts of interest, enabling them to interactively annotate system misbehavior by clicking on a timeline.
- A **Trace Analysis** algorithm that analyzes a smart home's history to detect potentially wrong or missing automations. It may infer an automation is missing if users often manually control a device to perform an action. Similarly, it may infer an automation is wrong if users often revert the corresponding action.
- A **Patch Synthesis** algorithm (Figure 3) that generates TAP patches based on intended TAP behaviors either explicitly specified by the user through the *History Visualization* interface or implied from the user's historical device usage by the *Trace Analysis* algorithm.
- A **Patch Behavior Visualization** interface (Figure 4) that shows users what would have happened in the history if a patch were applied.

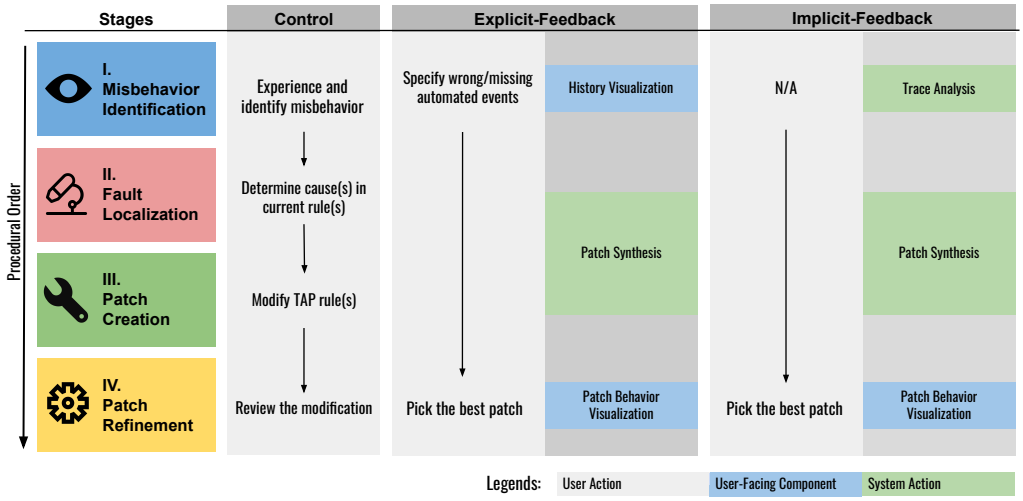


Fig. 1. Hypothesized cognitive stages of debugging and our proposed automated debugging workflows.

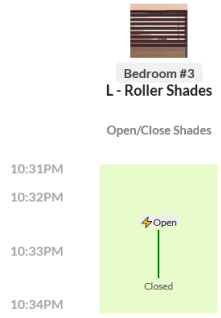


Fig. 2. History Visualization

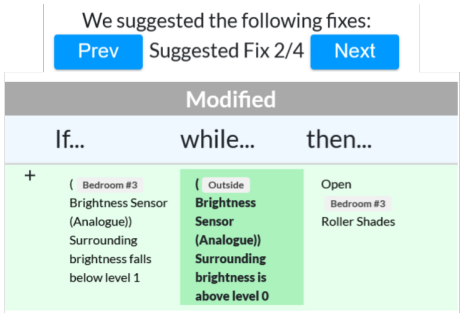


Fig. 3. Patch Synthesis Output

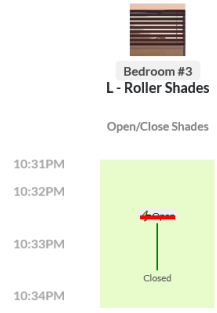


Fig. 4. Patch Behavior Visualization

We combine the above components into two debugging **workflows** as shown in Figure 1:

- **Explicit-Feedback**: (i) The *History Visualization* component receives from the user system misbehavior in the form of wrong or missing device automation(s); (ii) the *Patch Synthesis* component automatically generates TAP patches to fix the specified misbehavior; and (iii) the *Patch Behavior Visualization* component presents these patches for the user to select.
- **Implicit-Feedback**: (i) The *Trace Analysis* component infers system misbehavior implicitly based on the user's manual actions and reversions of automations; (ii) the *Patch Synthesis* component generates TAP patches accordingly; and (iii) the *Patch Behavior Visualization* component presents them to the user.

2) An empirical user study over the whole TAP debugging process, enabled by smart home simulation. To overcome the aforementioned challenges of time, cost, and participant privacy when conducting TAP studies in participants' homes,[1] and to enable more comparable deployments across participants, we extended the Home I/O [33] 3-D smart home simulator software (Figure 5, right) to support TAP. Specifically, we built a web application that connects client-side simulations in Home I/O to a server-side system for managing trigger-action

[1]The continued emergence of Covid-19 variants and related pandemic concerns make in-home studies even more challenging.

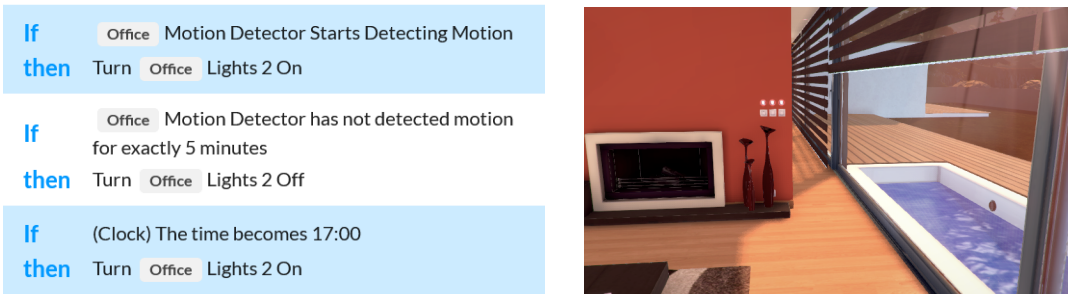


Fig. 5. We enabled trigger-action programs (left) to control the Home I/O smart home simulator (right).

programs, enabling TAP automation of all smart devices in the simulator. To support future research studies, we are open-sourcing our full TAP extension to the Home I/O simulator.[2]

In this study, we designed five TAP debugging tasks. In each task, a set of initial TAPs were pre-loaded in the simulated smart home. These TAPs (sets of if-then rules) were designed to be close to correct, yet occasionally exhibit problematic behavior. For example, rules might be set up to keep the room bright, but they sometimes would erroneously open the shade when users were asleep at night. We conducted remote interviews in which participants interacted with the simulator on their own computer while sharing their screen with a moderator. During these interviews, each participant's goal was to experience a misbehaving automation, diagnose the misbehavior, and eventually resolve the issue by modifying the TAP rules. In a between-subjects design, participants were assigned to either a control workflow where debugging was done manually without any tool assistance, or to one of our two novel workflows (Explicit-Feedback and Implicit-Feedback).

In total, we collected data from 30 participants across 84 hours of TAP debugging, split into multiple sessions per-participant to minimize fatigue. We were able to observe participants' approach to debugging problems, map out their mental processes, and identify the main obstacles that led to debugging failures. Through qualitative coding of the sessions, we observed a total of 16 obstacles that participants experienced across our four hypothesized stages of debugging. We also evaluated how well our two novel debugging workflows assisted participants in debugging TAPs. We discovered that our novel workflows (interfaces and underlying algorithms) led to significant improvements in participants' ultimate success in debugging TAPs and helped them avoid key obstacles.

2 TERMINOLOGY AND DEFINITIONS

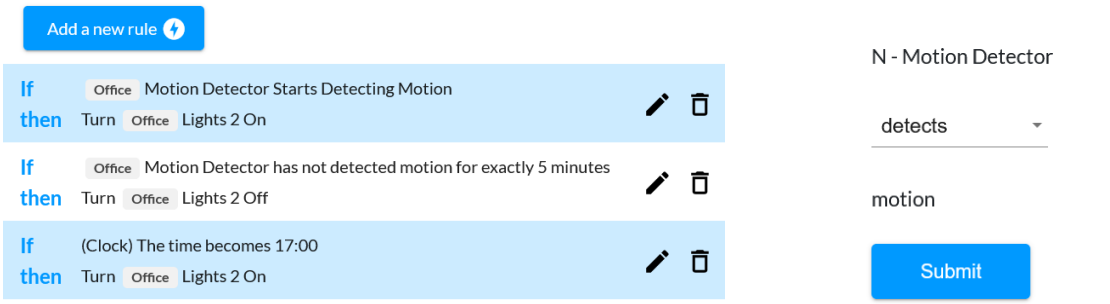
For precision when discussing TAP, we define the following concepts:

- An **event** represents a change in the status of a device or sensor. It happens only at exact moments. An example event would be "a light turns on at 10:03 pm on January 1st."
- A **trace** is a collection of non-automated and automated **events**. In this paper, a trace primarily refers to the history of smart devices in a smart home during a time period (e.g., yesterday from 8 AM to 1 PM).
- A **TAP rule** is an automation rule in the following form:

  "IF a **trigger** happens, WHILE **conditions** are true, THEN apply an **action**"

  – A **trigger** is a statement over events. For example, "the motion detector starts detecting motion" and "the temperature falls below 75° F" are both **triggers**.
  – A **condition** is a Boolean proposition over the status of device capabilities. For instance, "the temperature is below 75° F" is a **condition**. Note that **conditions** are optional for our version of TAP.

[2] https://github.com/UChicagoSUPERgroup/tapdebug.



(a) Rule page (b) Parameter configuration

Fig. 6. TAP management with the **Control** interface, representing current practice.

  - An **action** is a class of **events** with the same status change to the same device capability. For example, "turn on the light" is an action. An **event** is a single, concrete instance of an (abstract) **action**.

  If the **event** in a rule's trigger occurs, the system attempts to execute that rule, pending the **conditions**. If the system confirms the **conditions** are met, it will apply its **action** by sending a command to an actuator. Figure 6 displays three example TAP rules. We use **TAPs** to refer to a set of TAP rules.
- A **TAP patch** is a modification to a set of TAP rules. For example, if the original set of TAP rules should have happened under more cases, a possible patch would be to delete a condition from an existing rule.

3 NOVEL TAP DEBUGGING WORKFLOWS LEVERAGING USER FEEDBACK

To help users debug trigger-action programs, we have designed and implemented two end-to-end debugging workflows that offer automation and interface support at various debugging stages. We name these two workflows **Explicit-Feedback** and **Implicit-Feedback** based on the different ways that they take feedback about system misbehavior from users. In addition, we have also implemented a **Control** workflow that represents users' experience of manual TAP debugging without any tool support. In this section, we first summarize these workflows at a high level. We then provide further details about the interfaces and algorithms underpinning each.

3.1 Overview of Workflows

The **Control** workflow, representing current practice in deployed TAP systems, presents to users the set of TAP rules currently used by the smart home system, allowing users to make any changes they want to the rules in a web-app interface. Changes include deleting rules, adding new rules, and editing existing rules, as shown in Figure 6. This workflow offers no automated support and requires users to manually perform all debugging steps.

The **Explicit-Feedback** workflow leverages user's annotations of misbehaving automations to automatically suggest patches. As illustrated in Figure 1, users first *explicitly* annotate system misbehavior—what events in the history should or should not have occurred—through a history-visualization interface (see Section 3.2). Using that feedback, the system automatically synthesizes candidate patches, with each patch aiming to address a large fraction of the misbehavior. Section 4 details the patch synthesis algorithm. Finally, these candidate patches are presented, together with summary statistics and a patch-behavior visualization, to help users make informed decisions in patch selection and refinement. Section 3.4 details the presentation of patch candidates.

The **Implicit-Feedback** workflow differs from the Explicit-Feedback workflow only in terms of how it collects users' feedback about system misbehavior. Instead of requiring that specific misbehaviors be annotated explicitly, Implicit-Feedback automatically infers potential misbehaviors by analyzing users' behavior in the history, such as users manually actuating devices or reverting automations (e.g., turning off a light that has just automatically

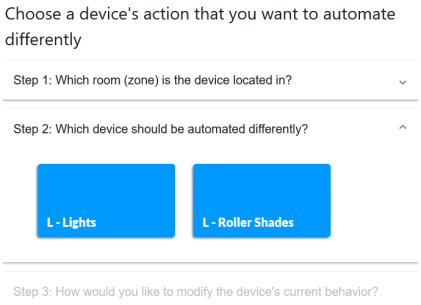


Fig. 7. Device selection (Explicit-Feedback)

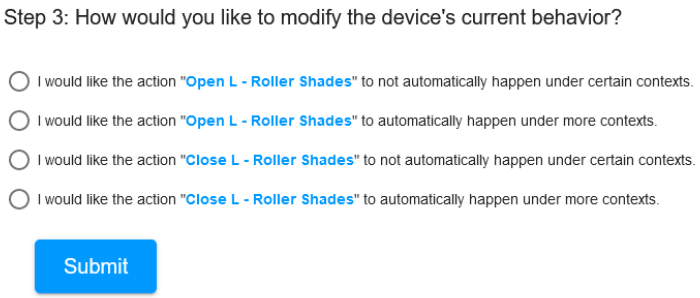


Fig. 8. Action and under-vs-over-automation selection (Explicit-Feedback)

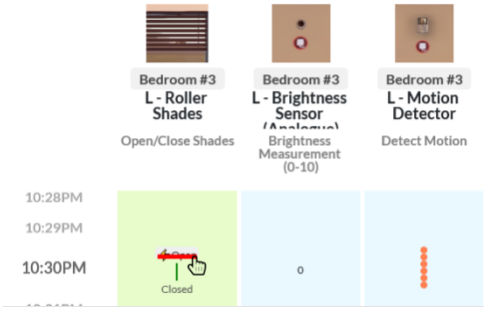


Fig. 9. Specifying that an event should not have happened, but did (Over-Automation)

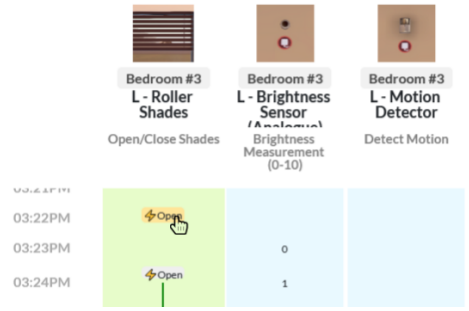


Fig. 10. Specifying that an event should have happened, but did not (Under-Automation)

turned on). Section 3.3 details how these inferences are made. Nonetheless, the user still specifies explicitly which device and associated action they believe to be misbehaving.

For the remainder of this paper, we will refer to the two types of misbehavior using the following two terms:

- **Under-Automation** is a false negative misbehavior where an automated event should have happened at a certain time, but did not (e.g., "the light should have been turned on at 7 PM last night").
- **Over-Automation** is a false positive misbehavior where an automated event did happen, but should not have happened then (e.g., "the light should not have turned on at 10 AM today, but did").

3.2 Explicit-Feedback: User Annotation of Automation Misbehavior

The Explicit-Feedback workflow asks users to explicitly specify system misbehavior, which will then be used for patch synthesis. Specifically, for every misbehavior case, users need to specify four pieces of information: (i) the problematic device, (ii) the problematic action, (iii) whether this is a case of over-automation or under-automation, and (iv) the time when the misbehavior occurred.

The bedroom light **should/shouldn't** have **turned on** at **7pm last night**.
(1) (3) (2) (4)

It would be overwhelming to ask users for all these details at once, identifying the single misbehaving device out of tens or hundreds of devices in a home and remembering the exact time of the misbehavior. Consequently, we have designed the following interface and workflow to help users specify these details step by step.

3.2.1 Device Selection. This workflow first asks users which device's automation is problematic. There might be many devices installed at people's smart home (e.g., 162 in the simulated smart home used in our study). To

not overwhelm users, the interface categorizes devices based on location (a specific room). Once users select a location, all devices installed at that location are displayed (Figure 7).

3.2.2 Action and Under-vs-Over-Automation Selection. Next, users are asked how they would like to modify the problematic device's automation. As shown in Figure 8, users are asked to pick from an exhaustive list that includes two options for every possible action of the device. These two options are (i) letting an action happen under more contexts (fixing under-automation), and (ii) letting it not happen under certain contexts (fixing over-automation). We intentionally combine the question about which action to modify and the question about under-automation versus over-automation into one question, as shown in Figure 8, because our pilot study found that asking them separately caused extra confusion. For example, when pilot participants noticed that the shade opened at an improper time, them often did not know whether to choose "open the shade" or "close the shade" as the action to modify. Pilot participants found the option "I would like 'open the shade' to not automatically happen under certain contexts" less confusing.

3.2.3 Time Point Identification. Users are then asked for time points when the specified under- or over-automation case(s) occurred. This is a challenging question for users as the misbehavior may have happened a while ago with many correct behaviors occurring subsequently. To help users, we developed an interface where users navigate a visualized event history of related smart devices and click to indicate time points of events to be cancelled or automated. For example, Figure 9 and Figure 10 show how users can click the time points of the exact events that should not have been automated and should have been automated, respectively. As shown in both screenshots, we offer the event history of not only the misbehaving device (e.g., Bedroom #3 Roller Shades in the figures), but also related devices that can offer contextual information to users and hence assist users' misbehavior pinpointing. We identified these related devices leveraging the open-sourced variable correlation analysis of Trace2TAP [42]. We allow users to specify multiple time-points where this specified device had under-automation (or over-automation) for the specified action so that users do not need to repeat this process.

3.3 Implicit-Feedback: Inferring Instances of Automation Misbehavior

The Implicit-Feedback workflow needs the same four pieces of information as the Explicit-Feedback workflow. Like Explicit-Feedback, Implicit-Feedback employs a user interface to elicit from users three pieces of information about misbehaviors — (i) the device, (ii) the action, and (iii) whether this is over- or under-automation. Different from Explicit-Feedback, the Implicit-Feedback workflow automatically infers the fourth piece of information — the time when the misbehavior occurred — by analyzing the trace, instead of asking the users. Going through the history visualization to find the times of misbehaviors is the most time-consuming part of the Explicit-Feedback workflow, which underpins our rationale for the Implicit-Feedback workflow. The inference is based on the intution that certain manual actions reflect what the user likes or dislikes. For example, if the user selected "I would like the action 'Open Roller Shades' to not automatically happen under certain contexts" in the interface shown in Figure 8 and the trace contained a record where the user manually closed the roller shade shortly after the shade was opened automatically at the time point T (Figure 11), the trace analysis can then infer that T is one of the cases of over-automation. Similarly, if the user selected "I would like the action 'Open Roller Shades' to automatically happen under more contexts" and the trace contained a record where the user manually opened the shade at T, the analysis can infer that T is one of the under-automation time points.

3.4 Explicit- and Implicit- Feedback Workflows: TAP Patch Presentation

After patch candidates are synthesized (see Section 4), they are carefully ranked and their prospective behaviors are visualized so that users can easily navigate among them and make an informed choice.

3.4.1 Patch ranking. Our ranking considers the following factors:

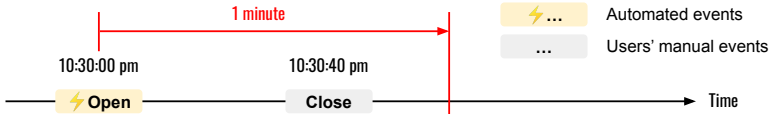


Fig. 11. An automated event that opened the roller shade is marked as a case of over-automation by Implicit-Feedback because it was manually reverted within one minute by the user.

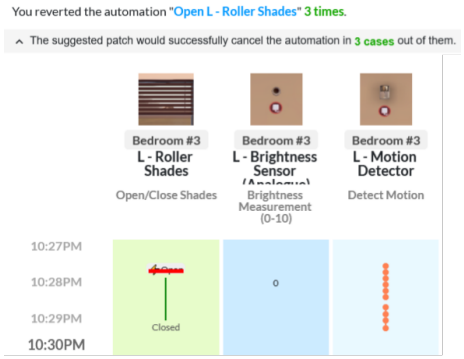


Fig. 12. Visualization to show a patch's behavior (fixing Over-Automation)

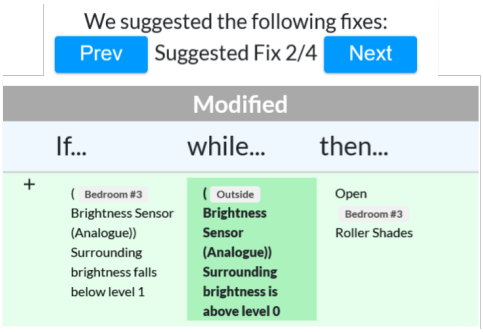


Fig. 13. A patch adding a new condition to an existing rule to (fixing Under-Automation)

- Number of fixed Under-Automations: We say an under-automation ("the target action should have automatically happened at time T") is fixed when the target action would happen between $T - \Delta_1$ and $T + \Delta_2$ if the patch were applied (Δ_1 and Δ_2 form a configurable time window). The more, the better for a patch.
- Number of fixed Over-Automations: We say an over-automation ("the automated event at T should have been cancelled") is fixed when the event would not happen if patch were applied. The more, the better.
- Number of cancelled events outside Over-Automation: When we apply a patch to the trace, it may stop automating an event that is not an over-automation. The less this occurs, the better.
- Number of newly automated events outside Under-Automation: When we apply a patch to the trace, it may automate the target action at moments that are far away from the under-automation time points. That is, assuming $T_1, ..., T_n$ are when the target action was under-automated in the trace, events newly automated outside $T_i - \Delta_1$ to $T_i + \Delta_2$ are undesirable. The less this occurs, the better.

We linearly combine these four numbers to get the final score of a TAP patch, presenting patches in descending order of score. We configure the weights among these four numbers as follows: $1, 1, -1, -0.25$. The last coefficient is smaller because we found in pilot studies that introducing extra automation beyond what users specified as under-automation is less of an issue and is sometimes even preferred. Patches accepted by users often led to extra automation at time points users forgot to mark as under-automation.

3.4.2 Behavior visualization. To help users compare patch candidates, we introduce a "stats and visualization" component in the patch presentation page. For each patch, we present the four metrics that we use to rank patches, as discussed above. Furthermore, when users click on a metric, a visualization of the corresponding events that would be automated or no longer automated by the patch is displayed. For example, Figure 12 shows that the patch, if applied, would not have automated a shade-open event at 10:28 PM.

4 AUTOMATICALLY SYNTHESIZING TAP PATCHES TO SUPPORT DEBUGGING

Our Explicit-Feedback and Implicit-Feedback workflows (Section 3) automatically generate TAP patches to fix over-automation and under-automation. Here, we formally define patch synthesis and detail our algorithm.

4.1 Problem Definition

The input to a patch synthesis problem includes:

- A **trace** containing the history of smart devices. It is a list of **event**s, either automated by TAP programs or manually conducted by human users, ordered chronologically.
- Misbehavior cases reported by users. If the misbehavior is about **over-automation**, the input includes events $E_1, ..., E_n$ in the trace, which all automate a specific **target action** of a device and should not have happened. If the misbehavior is about **under-automation**, the input includes a series of time points $T_1, .., T_n$, when a specific **target action** of a device should have occurred shortly before or after.

From these inputs, the synthesis algorithm aims to automatically generate the following output:

- A list of **TAP patch**es: each patch should fix over a threshold thd portion of misbehavior cases ($> thd \times n$). Here, thd is configurable. The higher it is, the stricter the synthesis algorithm would be and fewer patches would be generated. In our study, we used 0.3.

4.2 Intuition and Overview

A naive approach to finding all TAP patches that achieve our goal is to apply every possible modification to the current TAP rules, execute every resulting TAP program against the history trace, and see how many over- and under-automations could have been avoided by each patch. However, this approach is extremely time consuming, as the search space of all possible patches is gigantic. Instead of evaluating every possible patch, we use a symbolic approach. Specifically, our algorithm includes four components: (1) we design a symbolic representation for common TAP patches that can be applied to any existing set of TAP rules; (2) we execute the symbolic TAP program, which is the result of applying the symbolic patch to the existing TAP rules, against the history trace; (3) we formulate a symbolic representation of our patch-synthesis goal, like how many Over/Under-Automation cases in the history trace should be fixed; (4) we feed the trace execution result and the synthesis goal into a SAT solver, which will then generate all concrete TAP patches guaranteed to achieve our goal.

We use open-sourced tools Trace2TAP [42] and Z3[13] to handle components (2) and (4), respectively. We present our design for component (1) and (3) in the next two sub-sections. Note that the existing Trace2TAP tool is designed to automate manual actions by synthesizing new TAP rules when there are no existing rules. It offers a symbolic execution framework that shows symbolic TAP rules' behaviors in a trace. However, it cannot directly solve our TAP debugging problem, because, by design, it only attempts to add new TAP rules, but not to delete or modify existing rules. This is a fundamental limitation, because many debugging problems, particularly over-automation problems, can only be fixed by deleting or revising problematic rules but not by adding new rules. Furthermore, even for problems that can be fixed by adding new rules, like many under-automation problems, there are often simpler and hence more user-friendly patches accomplished by deleting conditions of existing rules or modifying parameters in an existing rule.

4.3 Symbolic TAP Patches

Before introducing different symbolic TAP patches in our system, we first give a brief overview of related concepts. A symbolic TAP patch is a patch containing **symbol**s that have not been assigned to a concrete value. As a result, the patch's behavior is non-deterministic (i.e., relying on values assigned to its symbols). We say a symbolic patch is concretized once all its symbols have been assigned concrete values. A symbol may appear as a parameter in a rule statement (e.g., "brightness is above level ***x***"), a device selector (i.e., which device is used in a rule depends

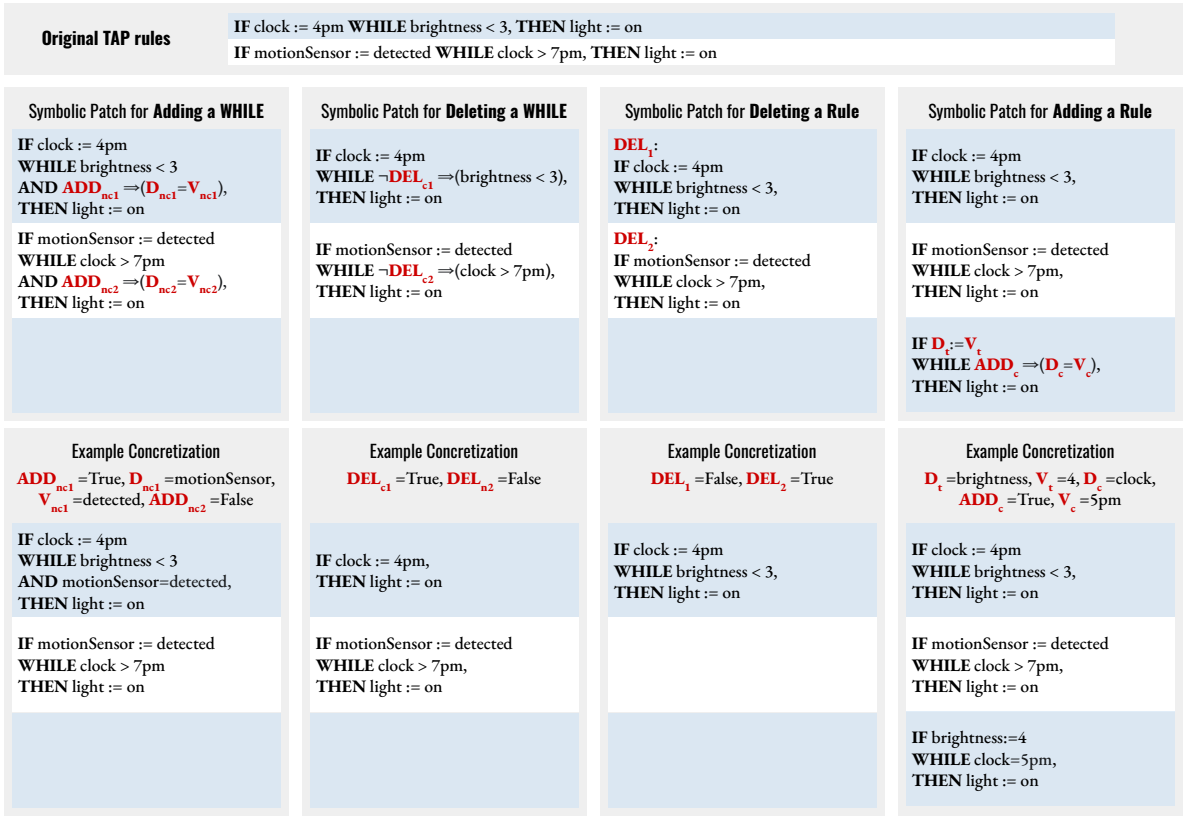


Fig. 14. Symbolic patches introduced over original TAP rules. In our real implementation, we also symbolize the comparators (=,<,>). For adding a new rule, the real implementation also supports having multiple WHILE conditions.

on the value of the symbol), an appearance flag for a WHILE statement (i.e., a condition only exists when the symbol is evaluated true), or an appearance flag for an entire rule.

Our system creates the following symbolic TAP patches over the existing TAP rules (Fig. 14).

- *Adding a new condition.* We introduce a symbolic patch that adds one condition to each existing TAP rule by attaching one symbolic WHILE condition to each one of them. We introduce the condition $D_{nci} = V_{nci}$ (the device D_{nci} is at value V_{nci}) only if ADD_{nci} is true.
- *Deleting conditions.* We can also put a DEL_{ci} symbol on each WHILE condition that already exists in the TAP rules. After the symbolic patch is applied, the WHILE condition i will only be checked if the associated DEL_{ci} is false.
- *Deleting rules.* To synthesize a symbolic patch that deletes rules, we can flag each existing rule (say, rule i) with a symbolic flag DEL_i. After applying the symbolic patch, rule i will be deleted if the flag DEL_i is true.
- *Adding a new rule.* Finally, we can introduce a symbolic new rule. Every component of this rule is symbolic, except for the action of this new rule: it is the concrete *target action* specified by users, which was either over-automated or under-automated in the past. In this new rule, both the IF statement and WHILE statement are symbolic.

4.4 Goal Expressions

Now, we can use the symbols used in our symbolic patch, denoted as *symbs* below, to represent whether a patch candidate has accomplished its debugging goal. As mentioned earlier, all the symbolic execution below is carried out in an existing symbolic TAP execution framework [42].

For each Over-Automation case specified by users, where event E should not have happened, we can symbolically re-execute part of the history trace where E.Trigger occurred - the event that triggered E - in a new system where the symbolic patch is applied and check whether E could be cancelled. We construct the following **Over-fix-expression**, F_{Over}:

$$F_{Over}([symbs...], E, trace) = \begin{cases} 1 & \text{if the event } E \text{ would be cancelled} \\ 0 & \text{if the event } E \text{ would not be cancelled} \end{cases}$$

For each Under-Automation case, where the target action should happen at around T, we can symbolically re-execute part of the history trace around time T (i.e., between $T - \Delta_1$ and $T + \Delta_2$; in this paper, we use $\Delta_1 = 10$ min, $\Delta_2 = 5$ min, following previous patch synthesis work[42].) in a new system where the symbolic patch is applied and check whether the target action would have been triggered. We construct a **Under-fix-expression**, F_{Under}:

$$F_{Under}([symbs...], T, trace) = \begin{cases} 1 & \text{if the target action would be triggered between } T - \Delta_1 \text{ and } T + \Delta_2 \\ 0 & \text{if the target action would not be triggered between } T - \Delta_1 \text{ and } T + \Delta_2 \end{cases}$$

Assume that users want to trigger the target action at a number of time points T_1, ..., T_n, or cancel a number of automated events E_1, ..., E_n, depending on what users selected in Fig. 8. Now we can express the expectation that a patch can fix over a threshold portion of misbehavior cases as the following goal expression:

$$\sum_{i=1}^{n} F_{Under}([symbs...], T_i, trace) \geq thd \times n \qquad \textbf{or} \qquad \sum_{i=1}^{n} F_{Over}([symbs...], E_i, trace) \geq thd \times n \tag{1}$$

Note that effects of modification unrelated to over- or under-automation are not considered in the goal expression 1. It is hard to automatically determine whether side effects comply to users' need. We consider them as soft factors in patch ranking (Section 3.4) instead of hard constraints.

4.5 Unified Patch Synthesis Workflow

Here, we put everything together. Given a set of existing TAP rules, four symbolic patches are generated over them, aiming to add a new condition to an existing rule, delete a condition from an existing rule, delete an existing rule, and add a new rule, respectively, as shown in Figure 14. Then, based on the misbehavior cases, we symbolically re-execute part(s) of the history trace with each of the symbolic patch applied. This process produces the goal expression for each symbolic patch, as described in Expression 1. We then send the goal expression to a SAT solver. If the goal expression is satisfiable, the SAT solver will return sets of concrete values for the symbols that achieve the goal. We concretize the symbolic patches with each set of values to get our TAP patches.

5 METHODOLOGY

In this section, we present the methods of our remote user study leveraging the Home I/O smart home simulator.

5.1 TAP and Smart Home Setting for Users

As mentioned in Section 1, participants in our user study experience TAP and smart homes in a simulator rather than a physical home setting. Instead of installing hundreds of smart devices at home and living with them for many days, participants in our study only need to (1) install and use a TAP-enabled smart home simulator and (2) run our TAP programming and debugging web application in their browser, as illustrated in Figure 15.

The TAP-enabled simulator is our extension of Home I/O. The original Home I/O is a game-like smart home simulator with 3D physical models developed by Real Games [33] (Figure 5). Users are able to move freely inside/outside a smart home and interact with a large number of smart devices in first person point of view. The

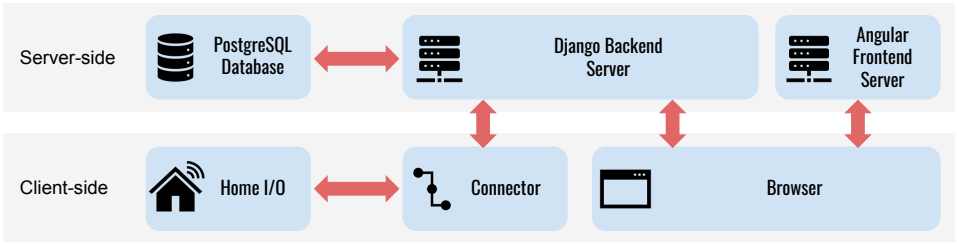


Fig. 15. Our framework that enables TAP for Home I/O

simulator also precisely models environmental factors such as sunlight and temperature in relation to time and seasons. Users can also fast forward time, or load pre-defined scenarios.

We need to extend Home I/O[33], as it does not support trigger-action programs. Fortunately, Home I/O allows external software to monitor and control all the devices inside the simulated home through a .NET 2.0 memory mapped file. Therefore, we developed a connector software that periodically reads the TAP rules stored in a backend Django server, parses the TAP rules, and constantly monitors and updates the status of all the devices in the simulated home based on the TAP rules.

Finally, our web application allows users to read and edit their trigger-action programs stored in the backend Django server, which can be fetched and used by the connector, as discussed above. In addition, our web application also receive traces of simulation from the connector and store them in the backend, enabling the two workflows.

As we can see, the above setting allows users to conduct our TAP debugging study in a more cost/effort efficient (without device purchasing and installation), time efficient (with fast-forwarding in the simulator), and fair (with every participant in the same home setting) way than in a physical setting.

5.2 User Debugging Tasks

We designed 6 tasks for each participant, with 1 of them being the tutorial task, as summarized in Table 1. Each task started with an initial goal and some TAP rule(s) that were set to achieve it. However, these TAP rules had some flaws as specified in the "problem" column in the table. The participant's goal was to modify the TAP rules so that the problem can be corrected while the initial goal would still be achieved.

We have carefully designed these tasks so that they collectively offer a good coverage of different task natures. As shown in the "Error type" column, some of our tasks aimed to let participants fix over-automation problems (i.e. imperfect rules wrongly triggered some actions), while others were about under-automation issues (i.e. imperfect rules failed to automate some actions). Furthermore, in some tasks, participants would notice the wrong or missing automated behavior right after it happened. In other tasks, however, participants would only be able to realize something went wrong later (e.g., in the next morning). This is illustrated in the "Discovered immediately" column in Table 1. The ideal modifications to fix the problems are shown in the "Ideal modification" column.

5.3 Pilot Studies

Before starting our formal interviews, we conducted 45 pilot interviews. These sessions helped us refine our workflow and protocol design in the following ways. To avoid participants' confusion, we merged the overautomation-vs-underautomation selection with the target action selection as discussed in Section 3.2. To reduce participants' cognitive load, we revised the tasks to be more concise. On the protocol side, we separated the study process into an onboarding phase and an interview phase to reduce researchers' waiting time. We also added a confirmation process to minimize no-shows. We also randomized the order of tasks to avoid confounds from learning effects.

Table 1. Tasks. Over-Automation and Under-Automation are shortened to "Over" and "Under."

ID	Problem	Error type	Discovered immediately	Ideal modification
0 (tutorial)	The garage door closes when people are under it.	Over	Yes	Add a WHILE condition
1	The light turns on even when it is bright.	Over	Yes	Add a WHILE condition
2	The blind opens when it's dark outside.	Over	Yes	Add a WHILE condition
3	The light turns on too late in winter.	Under	Yes	Change a parameter
4	The garage door is left open sometimes.	Under	No	Add a new rule
5	The entrance gate is left open at night sometimes.	Under	No	Add a new rule

5.4 Participants Recruitment and Assignment

We recruited participants from the United States on Prolific [32] in several rounds. In each round, we randomized the order of the 5 tasks, and recruited 3 participants to respectively use Control, Explicit-Feedback, and Implicit-Feedback workflows in a round-robin way. Every qualifying participants had a Windows machine and a mouse for running game-like smart home simulations. In our study, each participant went through two phases—an onboarding phase (Section 5.5.1) and an interview phase (Section 5.5.2).

5.5 Interview Process

5.5.1 Onboarding Phase. We surveyed participants on Qualtrics during this phase. First, we presented a consent form to each participant to make sure that all participants were over 18 years old, had proper computers/peripherals for the simulation, and agreed to participate in the study. Next, each participant went through a tutorial on trigger-action programming (TAP), which taught key concepts such as the TAP syntax and the difference between state statements used in IF triggers ("the window is open") and event statements used in WHILE conditions ("the window becomes open"). We then asked each participant simple questions about TAP as an attention check. Subsequently, we presented participants instructions on installing Home I/O and the Connector. At the end of the survey, participants answered questions on demographics.

5.5.2 Interview Phase. By this point, we assumed that participants had successfully installed the simulator and the connector. After giving us their consent on screen and speech recording, participants shared their screens and we started the recording and the interview. During the interview, we first offered a brief tutorial on the simulator. During this tutorial, participants familiarized themselves with operations in the simulator, such as moving, interacting with devices, reading the mini map, and controlling the time flow. Subsequently, participants went into the main part of the study: task solving.

Participants were asked to solve Task 1–5 (Table 1), with the order randomized, after we demonstrated the whole process by solving the tutorial task, Task 0, using their assigned debugging workflow. For each task, we showed them the goal, the existing rules, and the problem to be corrected in a summarized version without any specific information on causes and contexts. With the previously mentioned information and possibly some procedural clarifications from the researcher, participants then went through some daily routines related to the rules in the simulator, experiencing both desired and undesired behavior of the current rules by themselves. Afterwards, participants were directed to their assigned TAP debugging workflow. We then asked participants to find out what went wrong with the rules and fix them with the assigned workflow tool, recording whether they submitted a correct answer. In this process, we only answered participants' questions about interface usage, but not ones related to tasks, TAP logics, or patch behaviors. Finally, after each task, we asked participants a set of questions regarding their debugging experience — for example, whether they were able to identify which part in

Table 2. Distribution of participants' correctness rate in each task with the 3 interfaces. The x-axis is tasks. The y-axis is the number of participants.

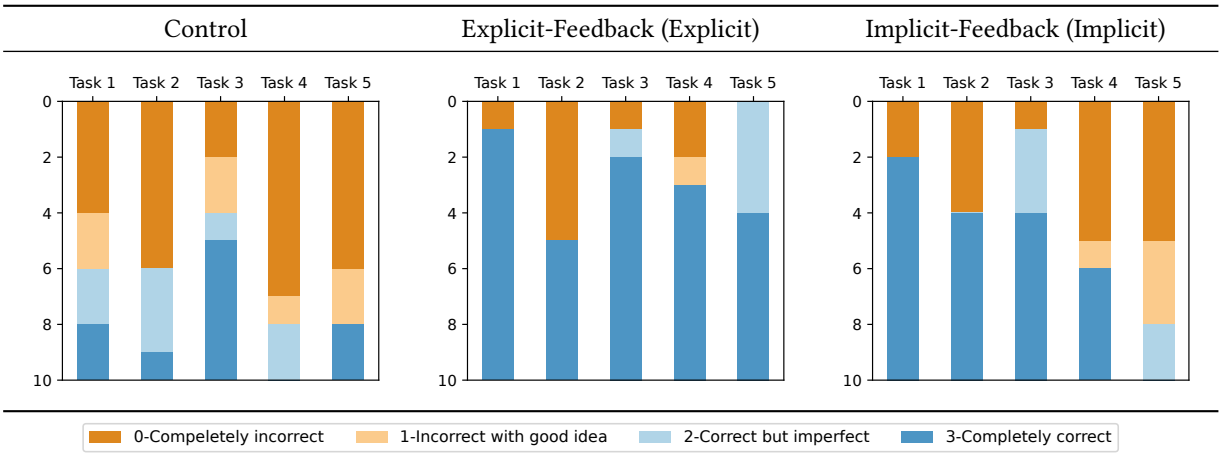


Table 3. P-values for statistical tests we performed: the smaller the p-value is, the more evidence we have in favor of the alternative hypothesis. Hypothesis-1 was tested using the Kruskal-Wallis test adjusted with Benjamini-Hochberg method. Hypothesis 2 and 3 were tested using Mann-Whitney U test. Significant p-values (< 0.05) are bolded. N/A: The Mann-Whitney U test was not conducted when there was no statistical significance for hypothesis 1.

Alternative Hypothesis	Task 1	Task 2	Task 3	Task 4	Task 5
1. Results of three interfaces are from different distributions.	**0.0223**	0.3432	0.3433	**0.0358**	**0.0042**
2. Explicit-participants performed better than Control-participants	**0.0096**	N/A	N/A	**0.0095**	**0.0099**
3. Implicit-participants performed better than Control-participants	**0.0328**	N/A	N/A	0.1980	0.9668

the rule(s) were wrong before going into this workflow, whether they understood the task, and whether they were confident in the final patch they submitted. Afterwards, participants completed the System Usability Scale for the workflow.

We encouraged participants to manually revert some wrong actions of automated devices in the simulator as what they would do in real life (e.g., turning off a light that has been mistakenly turned on). This is crucial as the Implicit workflow relies on users' interactions with sensors and devices to synthesize patches.

5.6 Coding of Results

From interview recordings, two researchers independently coded up the correctness scores of all 150 sessions (3 workflows × 10 participants × 5 tasks) from "0-completely incorrect" to "3-completely correct", meanwhile identifying obstacles faced by the participants. Note that while the correctness scores usually have definite, referable values, the obstacles can be ambiguous and inferred. As a result, we identify obstacles participants encountered in a precise but not complete manner. These two researchers resolved each conflict by reviewing and discussing the corresponding recording. Cohen's Kappa was calculated between the correctness scores. For reference, we have attached our code book in the Appendix 1 - Code Book.

6 RESULTS

In this section, we first give an overall review of the performances of 3 interfaces (Section 6.1). The next section (Section 6.2) details the results of Control-group participants and the typical obstacles they encountered. Finally, we report the performances of non-control interfaces and evaluate their strengths and weaknesses (Section 6.3).

6.1 Overall Correctness by Workflow

Two researchers independently scored the correctness of the 150 sessions. with a Cohen's Kappa of $\kappa = 0.86$. The distributions of the final correctness scores are shown in Table 2; identified obstacles are summarized in Fig. 16. Here, we elaborate on several general findings.

When automated tools were absent, participants struggled to solve our tasks. As shown in the "Control" row of Table 2, except for Task 3, the control-group participants performed poorly in every task, where only 0–2 out of the 10 participants scored "3-completely correct". In fact, in Task 2, 4, and 5, the majority of the control-group participants scored "0-completely incorrect". Even in the simplest task, Task 3, only half of the participants scored "3-completely correct", while 4 of the participants scored "0-completely incorrect" or "1-incorrect with good idea".

Generally, participants in the Explicit-Feedback and Implicit-Feedback groups were able to achieve better correctness scores than their counterparts in the Control group. As shown in Table 2, compared with the control group, the Explicit-Feedback group had more participants completely solve the problem (i.e., scoring "3") and fewer participants completely fail the problem (i.e., scoring "0") in *every* task. The Implicit-Feedback group also performed better than the Control group: it had more participants completely solve the problem in every task, except for Task 5, and fewer participants completely fail the problem in every task than that in the Control group.

Such supremacy in correctness is not homogeneous but interestingly differentiated from task to task. For each task, we performed statistical test, with the details shown in Table 3, and discovered that:

- In Task 2 and 3, Explicit-Feedback and Implicit-Feedback interfaces performed better than Control, but not with statistical significance, as indicated by the omnibus test (Kruskal-Wallis) result shown in Table 3. For task 3, no matter which interface was used, participants generally performed well, as this task was simple, only requiring the participants to fix a time parameter in an existing rule (i.e., "IF clock turns ~~6:20pm~~4:00pm, THEN turn on the light"). For task 2, there was a different story. Participants in the Control group struggled to come up with the right patch—only 1 participant figured it out. Both automation interfaces were able to automatically synthesize the right patch. However, only 50–60% of the participants were able to pick the right patch out of multiple patch candidates due to mental obstacles that will be discussed in 6.3.2, which diminished the advantage of the two automation interfaces.
- In Task 1, both Explicit-Feedback and Implicit-Feedback interface performed better than Control with statistical significance.
- In Task 4 and 5, Explicit-Feedback performed better than Control with statistical significance, and yet Implicit-Feedback did not. What differentiates these two tasks from the other tasks was that home users were *not* on site (e.g., they were sleeping) and hence did *not* immediately discover the home system's misbehavior, as shown in Table 1. As a result, without the immediate manual feedback from home users, Implicit-Feedback interface only offered limited help.

Finally, all three interfaces received similar usability scores, although the Implicit-Feedback interface received the least questions from participants. SUS scores were generated per participant. If a participant took multiple interviews to finish all tasks, we calculated the average score across interviews for her. Control, Explicit-Feedback and Implicit-Feedback respectively scored a mean SUS of 71.7, 74.9 and 72.6 among participants. The difference was not significant (F statistic: 0.0768, p-value: 0.926 from one-way ANOVA). Although the SUS scores were

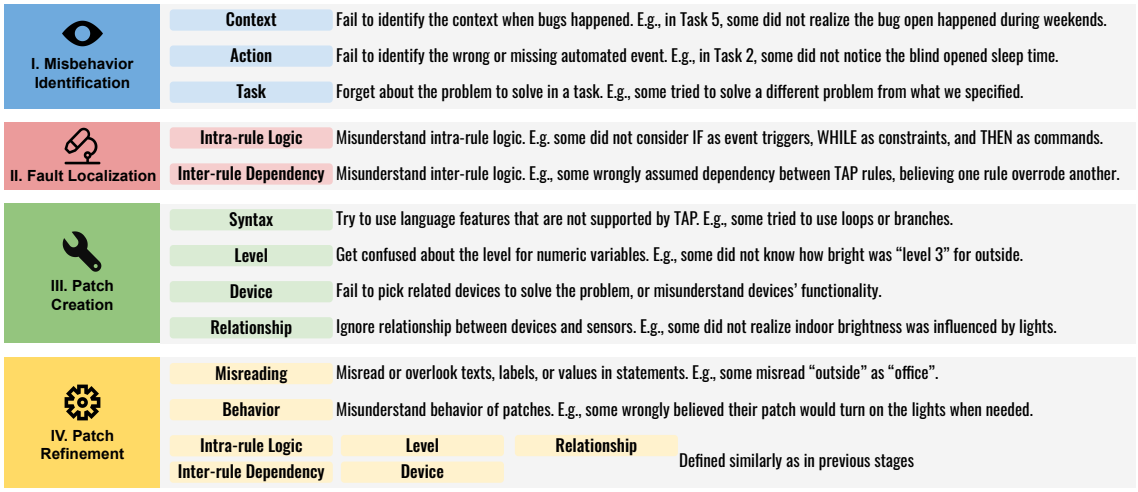


Fig. 16. Obstacles end-users face in debugging TAP

similar, there were the fewest sessions where participants needed clarification on the interface from us in the Implicit group (Control: 40 questions out of 50 sessions, Explicit: 48/50, Implicit: 18/50). With the Explicit-Feedback interface, participants spent more time in average finishing each task than with Control and Implicit-Feedback (Control: 4m54s, Explicit: 10m57s, Implicit: 5m57s) due to the extra step asking for TAP behavioral feedback.

6.2 Obstacles Faced in Manual Debugging

To understand what exact obstacles are encountered by end-users during their manual debugging, we watched the interview recordings and summarized typical obstacles participants faced (criteria shown by our code book in the appendix). The full list of obstacles and their definitions are shown in Fig. 16, and the frequencies of obstacles' occurrences are shown in Fig. 17. In this section, we discuss some of the major obstacles.

Note that, more than one obstacle could be encountered for a participant when he/she tackled a task, in which case, the different obstacles might contribute differently to the potential debugging failure. Also note that, a user who encountered an obstacle might still be able to correctly finish a debugging task, although this was very rare without automated tool support, only occurring in 2 task sessions (one in Task 1 showing the "Level" obstacle, another in Task 3 showing the "Level" and "Syntax" obstacles) in our study.

6.2.1 Fault-Localization::Intra-rule Logic. To localize the specific rule or rule component that has caused the system misbehavior, end users need to accurately understand the rules. During this process, it was common for participants to inaccurately comprehend the meanings of IF, WHILE and THEN statements, supposedly because these terms are polysemous in daily usages. In the 50 sessions with control-group participants (5 tasks × 10 participants), such confusion was revealed in 21 sessions and all of these 21 sessions ended up with debugging failures (correctness score < 3). These confusions primarily fall into the following categories:

- **Perceiving IF statements as qualifiers on states, instead of on events.** Although our TAP tutorial taught every participant that the IF statement describes an event whose occurrence may trigger an action, many participants mistakenly interpret the IF statement as describing a system state that is repeatedly checked . For example, a statement like "IF brightness goes below level 3" only takes effect at the moment when the measurement of a brightness sensor dips from above 3 to below 3. However, several participants think that it consistently takes effect as long as the sensor reports a brightness level below 3.

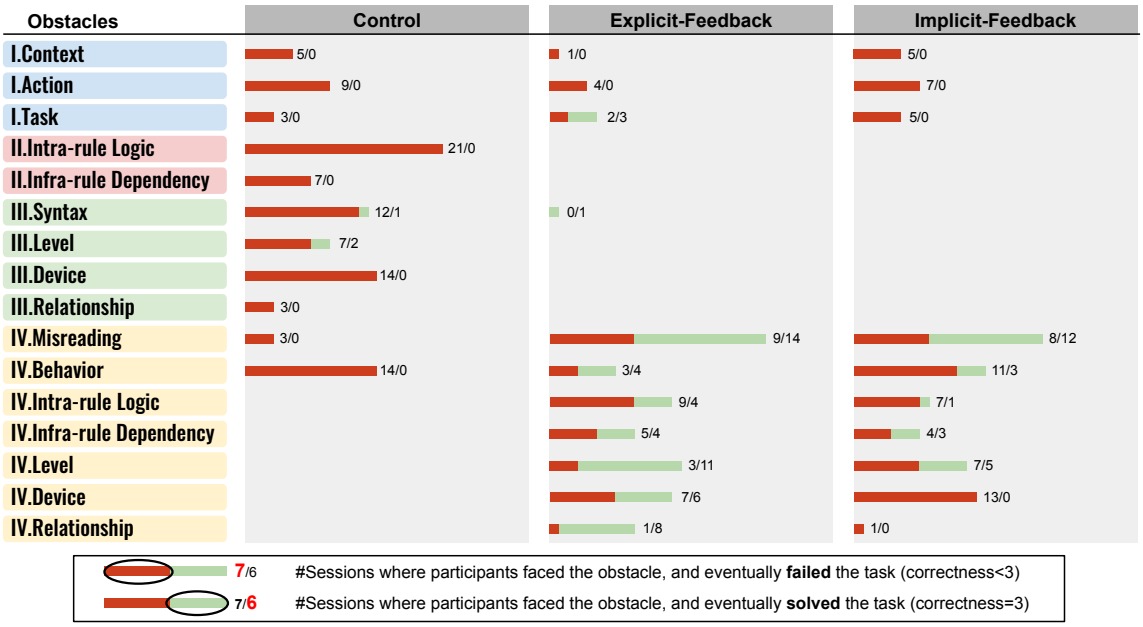


Fig. 17. Frequency of obstacles' occurrence: in how many sessions (among 50 = 5 tasks × 10 participants for each interface) did each type of obstacles showed up. A more detailed task-specific table can be found in Appendix 2.

- **Perceiving WHILE statements as qualifiers on events, instead of states.** WHILE statements restrict the triggering of a rule—even if the event in the IF statement occurs, the rule is not triggered if the state in the WHILE statement is not held. However, some participants misunderstood WHILE to be an alias of IF. For instance, some participants added "WHILE the clock is 10pm" to an existing rule and thought that the rule would get additionally triggered at 10pm. In reality, this extra WHILE statement caused the rule to almost never take effect, as when the IF event occurs the clock is almost never 10pm.
- **Perceiving IF and WHILE statements as describing actions to take.** The action to take by a TAP rule is only what inside the THEN statement, but some participants got this wrong. For example, one participant created a rule "IF clock turns 11pm, WHILE the gate is closed, THEN arm the central alarm", and told us that at 11 pm, the system will not only arm the central alarm, but also close the gate.

6.2.2 Fault-Localization::Inter-rule Dependency. Different rules in a TAP program are triggered independently. However, some participants assumed non-existent relationships among rules. With such a wrong understanding, they failed to accurately identify the misbehavior-inducing fault in the existing TAP program. This showed up in 7 sessions with control-group participants in the following ways.

- **Assuming that two rules with opposite actions could completely offset each other.** In both Task 1 and Task 2, the faults of the TAP programs are that the triggering conditions of some rules were too loose, causing excessive rule action. Unfortunately, some participants believed that the faults were the missing of rules with opposite actions. This belief led them to add new rules with opposite actions from those in existing TAP rules. Their debugging attempts failed, as it is very difficult to coordinate two rules so that one can precisely offset the other at the right moment. For example, in Task 2, some participants believed a rule of "IF clock turns 10pm, THEN close the blind" would ensure that the original "IF brightness falls below level 1, THEN open the blind" would not happen after 10pm, which is wrong.

- **Assuming that rules are triggered sequentially.** When there are multiple TAP rules listed, some participants believed that the later listed rules would only be triggered after the earlier listed rules were executed. We hypothesize that this confusion came from their experience with programming languages where code was interpreted sequentially.
- **Assuming that a WHILE statement constrains all TAP rules.** Some participants believed that all rules would be constrained by a WHILE statement newly attached to a single rule.

6.2.3 Patch-Creation::Syntax. The simplicity of trigger-action programming comes at the cost of missing some general programming language features, which puzzled some participants, especially the expert ones (i.e., who had at least some programming experience), during patch creation. Generally, while expert participants performed better than non-expert participants (Spearman correlation = 0.275, p-value = 0.0530 between experience and correctness score), the former encountered a bigger "Syntax" obstacle, projecting their knowledge of general programming languages onto TAP. In Task 4, for example, 7 participants completely gave up on solving this task, as they were determined to use *loop*s, a common feature in general programming languages, to periodically check the status of an infrared sensor and yet *loop* is not supported in TAP. Notably, 6 among them were expert users. In this regard, we speculatively suggest that users' thinking sets may impede them from realizing a simpler, supported solution - "IF infrared-sensor stops detecting something, THEN close the garage door" - and this phenomenon seemed considerably more prominent among experts.

6.2.4 Patch-Creation::Device. In 14 of the 50 control-group sessions, participants did not demonstrate good understandings of related devices (e.g., sensors) - that is, the devices they suggested, either explicitly or implicitly, could not possibly compose a rule to solve the problem. Specifically, this obstacle typically comes in two forms.

- **Unawareness of existences:** Some participants failed to realize that there existed sensors that would fit their need. For example, the solution to Task 2 is to have one of the rules to not be triggered when it is dark outside, including during the night time. 4 out of 10 participants were not aware that there was a brightness sensor outside, and hence failed to solve the problem.
- **Misunderstanding of functions:** Participants may also misinterpret the functions of relevant sensors even when they have noticed the existences of relevant sensors. In Task 4 and 5, participants often mistook motion detectors as sensors for presence instead of movements.

6.3 Our Novel Workflows' Impact on TAP Debugging

We have already shown in Section 6.1 that the use of our two automated debugging workflows - Explicit-Feedback and Implicit-Feedback - led to more debugging success in most tasks. In this sub-section, we discuss their advantages (Section 6.3.1) and limitations (Section 6.3.2) against the Control workflow, as well as the comparison between these two workflows (Section 6.3.3) in details.

6.3.1 General Advantages. Based on our interview with the participants, the main advantage of these two automated debugging workflows is that they made obstacles in **Stage II: Fault Localization** and **Stage III: Patch Creation** less consequential to the debugging result, comparing with the Control workflow. As per the nature of these two non-Control workflows, participants only needed to provide to-be-modified target action and (only for Explicit-Feedback) specify the exact wrong behavior to have the system generating candidate patches for them (Fig. 1). In terms of the obstacle stages, the system bridged **Stage I: Misbehavior Identification** and **Stage IV: Patch refinement** for participants and handled the other two stages on its own with the patch synthesis algorithm (Section 4). As a result, obstacles originally in **State II and III** such as Intra-rule logic, Level and Device were now shifted to the later stage, **Stage IV: Refine the patches** (Fig. 16, 17). This stage shift is crucial: by effectively changing participants' role from patch creator to reviewer, Explicit-Feedback and Implicit-Feedback workflows diluted the influence of these obstacles on the correctness of final modifications. For example, as

shown in Fig. 17, among all 21 Control-group sessions in which participants expressed confusion Intra-rule Logic, none had the participant correctly fix the bug; in contrast, this proportion skyrocketed to 14/23 and 12/20 with the use of Explicit-Feedback and Implicit-Feedback interfaces, respectively. Similar trend was seen for other obstacles such as Level, Device and Infra-rule Dependency. We explain this distinction with our use of the patch synthesis algorithm - our system automatically configured patches with correct TAP logic and reasonable parameters from the history (trace). As a result, using the two feedback-based workflows, participants were able to select correct rules even without rigorous understanding of devices, sensors, and TAP logic.

6.3.2 *Limitations of our automated debugging workflows.*

Explicit-Feedback and Implicit-Feedback workflows merely delayed and alleviated - not eliminated - these obstacles. From our interactions with the participants, we saw that confusions could make participants reject the correct patches or select wrong ones: participants could still fail with Explicit-Feedback or Implicit-Feedback (Fig. 17). Among all 100 sessions performed with non-control workflows, 10 of them had participants rejecting all correct patches synthesized by the system. For example, a participant who thought that additional WHILE statements would cause the rule to trigger more (Intra-rule Logic obstacle) rejected a correct patch that prevented the blind from wrongly opening because she thought it would do the opposite.

Automated debugging workflows might increase users' probabilities of misreading. The obstacle of "Misreading," as illustrated in **Stage IV: Patch Refinement** in Figure 16, refers to participants' linguistic errors during their comprehension of statements on the debugging interfaces, which included but were not limited to misreading device locations (e.g., taking office brightness as outside brightness), comparators (e.g., taking above as below), or action verbs (e.g. mistaking open as close). Interestingly, this confusion seemed to be a major obstacle only for participants in the two non-control groups: statistically, only 1 session in the Control group had demonstrated any significant sign of misreading, whereas Explicit-Feedback and Implicit-Feedback each had 28 and 18 instances. To account for this phenomenon, we hypothesize that users are more prone to misreading statements that are not configured by themselves: the two feedback-based workflows both had the system instead of the participants generate TAP rules, so reasonably participants would have higher chances of misreading.

Users could be confused about the over-automation vs under-automation selection. The Explicit-Feedback and Implicit-Feedback workflows require users to specify whether the misbehavior is an over-automation or under-automation issue (Fig.8). For example, if the issue was about roller shade opening under undesired contexts (Task 1), it would be an over-automation issue about shade-open (option 1 in Fig. 8). However, some participants failed to make this connection. Wrong selections had been made in 9/50 and 6/50 sessions with Explicit-Feedback and Implicit-Feedback. What makes this confusion fatal and thus more crucial is that both workflows would not be able to suggest good fixes without such information from users.

Our patch behavior visualization had limited success. When showing candidate patches with the two automated debugging workflows, we recreated histories assuming that the patch was originally in place (Fig. 12). We made this visualization information-comprehensive (i.e., encompassing all introduced behaviors and all potentially related devices and sensors) expecting that more details would generate more value to end users. However, only in 9 out of all 100 sessions, participants consulted the behavior visualization section to make decisions, despite that the interviewers had introduced its potential usages beforehand in the tutorial.

6.3.3 *Comparison between Explicit-Feedback and Implicit-Feedback.*

The Explicit-Feedback workflow reduces the "Action" and "Context" obstacles in ***Stage I: Misbehavior Identification****.* With Explicit-Feedback, participants were required to identify wrong automated behaviors in the visualized smart home's history (Fig. 9,10). Although this encouraged participants to talk more about the context

and exact wrong automated events for the misbehavior, participants showed fewer signs of facing the "Context" and "Action" obstacle than these in Control and Implicit-Feedback. We hypothesize that explicitly reading about history of related devices helps end-users better understand contexts and wrong events about the misbehavior.

The Implicit-Feedback requires users to provide instant feedback in order to generate patches. The Implicit-Feedback offers the advantage of requiring the least amount of inputs from users at the cost of relying on participants' instant manual reactions to wrong or missing automated events in order to identify misbehavior to fix. As a side effect, the Implicit-Feedback workflow would likely fail to address when participants were not able to give instant feedback of the wrong behavior. In Task 4 and 5, participants were only able to observe the consequences of the wrong or missing automation long after it happened: for example, in Task 5, participants would not realize the door had been left open throughout nighttime until the next morning. Naturally, participants were able to give no instant manual feedback in these two tasks. Indeed, the correctness result for the 2 tasks with Implicit-Feedback was not good: for Task 4 and Task 5, only 5 and 0 out of 10 sessions each had any correct patch showed up. While the data for Task 4 (5 out of 10 sessions) did not seem persuasive enough, there was a noteworthy reason on why the correct patch showed up in these cases: we analyzed the traces of Task 4 and discovered that these 5 successes were in fact due to some unrelated routines, not the one to be modified. To conclude, when users could not immediately apply manual feedback, while correct patches may still show up by chance, it would be better to use the Explicit-Feedback workflow instead.

The learning curve is lower with Implicit-Feedback than with Explicit-Feedback. Although the Explicit-Feedback workflow might be more effective in more tasks (Table 2), it also seems to require more effort from the users, by requiring the users to carefully interact with simulation histories to provide explicit feedback. In contrast, the Implicit-Feedback workflow enables users to be "just one click away" from synthesized results. This comparison between workflows led us to believe that the Implicit-Feedback workflow requires less tutorial for end users than the Explicit-Feedback workflow. Results from our interviews also support this hypothesis. During the interview, when participants expressed non-TAP related confusions, we gave some hints and tutorials. In 48/50 sessions with Explicit-Feedback had participants receive clarifications or hints about the interface, whereas this number went down to 18 for the Implicit-Feedback group. (The number was 40 for the Control group.) The part that required the most additional clarification in the Explicit-Feedback interface was the manual behavioral feedback page (Fig. 9,10). We often needed to illustrate how to suggest a new automated event or cancel an existing event.

7 RELATED WORK

In this section, we summarize related work.

7.1 Program Debugging

Studies to understand the debugging procedure for computer programs have been done since the start of the information age. Gould [15] recruited experienced programmers to debug FORTRAN programs and compared the performance between different tasks and between programmers with different levels of experience. Another previous work [36] investigated the debugging process on COBOL programs of 16 novice or expert programmers relative to a cognitive debugging model established from literature. The model involved hypothesis generation, hypothesis evaluation, error confirmation, code localization, and so on. Xu et al. [39] adapted Bloom's taxonomy[2] in the field of education to program debugging based on an observational case study of expert programmers debugging an information system that consists of a web server and a relational database management system. In more recent years, researchers have also studied the debugging process of end users. Grigoreanu et al. [16] conducted an empirical study on end users' sensemaking about spreadsheet correctness and derived mental models of end-user debugging on spreadsheets. Our empirical study complements these existing works, since

we observed debugging behaviors of end-users instead of programmers and explored unique obstacles faced in debugging trigger-action programs.

7.2 TAP Misunderstanding Classification

Prior works have classified bugs in trigger-action programs. By asking online participants to interpret and create TAPs, Huang et al. [19] systematically analyzed users' confusion towards concepts and semantics in TAP such as event trigger vs state trigger. The work from Brackenbury et al. [3] identified 10 classes of TAP bugs caused by control flow confusion, timing confusion, and inaccurate user expectations. Brackenbury et al. further tested if online participants could correctly understand behavior of TAP rules in the presence of these bugs. Another work by Palekar et al. [31] classified errors users could make in TAP into 9 categories based on previous works and examined whether current TAP platforms discouraged such errors. Different from prior works, our work employed end-to-end TAP debugging user studies to understand users' mental obstacles during the whole process of debugging, which goes beyond just TAP rule interpretation and TAP rule writing: while TAP-interpretation-related obstacles like Intra-rule Logic have been found in both prior works and our study, our study identifies many other types of obstacles related to misbehavior understanding (e.g. Action), device understanding (e.g., Level and Device), and so forth, as shown in Fig. 16. Furthermore, our study shows how these obstacles influence users' ability to successfully modify TAP rules and fix the misbehavior of a smart home system.

7.3 Bug Detection for Smart Home Automation

Much work has been done to automatically detect bugs in trigger-action programs using static analysis and model checking. Every bug detection technique focuses on bugs of a particular type, such as violations to a set of device-specific safety properties [1, 6], violations to a set of system properties manually defined by users in the form of "should not happen" or linear temporal logic (LTL) [4, 25, 41], TAP rule interaction vulnerabilities, such as action duplication and action conflict [37, 40], rule conflicting and rule chaining [8, 10, 14, 18, 21, 23, 38].

By design, bug detection and debugging complement each other, with this work focusing on the latter. A debugging process starts from a misbehavior of a device reported by users, and ends at the misbehavior getting fixed through changes to the TAP program. This misbehavior could be caused by various reasons: a bug of any type in the TAP program, a shift in the preference of the users, a change of behavior of the users or the environment, and others. Better bug detection can reduce but not eliminate system misbehavior.

7.4 Program Synthesis

Recently, researchers have started trying to synthesize trigger-action programs. AutoTap [41] synthesized TAP programs/patches that satisfied user-specified LTL safety properties by analyzing transition systems. Trace2TAP used symbolic execution and SAT-solving to generate TAPs from users' behaviors in the history [42]. RuleSelector [34] identified most promising TAP rules for users with data mining on their history. Corno et al. [11] implemented a semantic-based and conversational recommendation algorithm to generate TAP rules through natural-language dialogues. Compared to these works that only synthesize new TAP rules [11, 34, 42] or patches based on LTL properties [41], our work makes program synthesis foray into the field of TAP debugging - we synthesize TAP patches that can fix misbehavior cases in the history. The different goals led to different techniques used by us and previous work. For example, the transition-system technique used by previous patch synthesis work AutoTap [41] ties closely with LTL properties and hence does not work in our case.

7.5 Smart Home Visualization

Our patch behavior visualization component is inspired by prior works on TAP visualization. Mennicken et al. [27] built an interactive calendar-like interface that visualized behaviors of smart devices. The interface was evaluated with a field study in 2 commercial smart homes. Zhang et al. [42] built a timeline visualization showing new automated events introduced by synthesized new rules. Castelli et al. [5] developed a dashboard for smart home users that presented current and history status of devices. Corno et al. [10, 12] and De Russis et al. [14] allowed users to simulate behaviors of their TAPs step-by-step. Coppers et al.[9] predicted and simulated potential future behaviors of TAPs for users. Zhao et al.[44] helped users to distinguish between two set of TAPs by visualizing their differences. Compared with these prior works, our work goes beyond visualizing behaviors of specific TAP rules - in the Explicit-Feedback workflow, it uses the history visualization of the smart home as an interactive interface to collect system misbehavior experienced by users.

8 THREATS TO VALIDITY

We illustrate this work's limitations from the following aspects.

Data Collection. First, we only passively collected evidence of mental obstacles based on participants' conversations with us. As a result, how much data was collectable relied on the extent to which participants exposed their confusion; our identification of obstacles was thus precise but not complete. Second, we attempted to understand users' tool preferences only through reported usability. Other types of usability data and ways to gather them can be explored: particularly, it would be crucial to understand users' tool preferences regardless of performance.

System Development. On the algorithmic side, we have only considered several major types of patches (Fig. 13) to present usage of our tools. There might be debugging tasks that can be better fixed with types of patches we did not consider (e.g., modifying existing WHILE conditions). Beyond that, some of the obstacles identified in this paper may be lowered by rendering a clearer interface design. For example, one may possibly lessen users' likelihood to misread statements by emphasizing the easily confused parts (e.g., Infrared **1** vs Infrared **2**). Nevertheless, we also note that this limitation does not affect the majority of the obstacles investigated and conclusions made.

Study Design. Regarding the procedure, our protocol guided participants through pre-set routines, with the goal to expose our pre-designed misbehaviors. Tasks tested under this protocol may not represent all TAP debugging problems, and users may perform differently if they spontaneously identify misbehaviors instead. Also, we set up our study in a simulator. While this design has yielded several notable advantages, participants may have difficulties adapting to virtual settings, and their experience of events (especially misbehavior) may differ from the reality. Such disparities may weaken the generalizability of our findings.

9 DISCUSSION AND FUTURE WORK

This work focuses on TAP debugging, a process that is frequently conducted by TAP users but has not been well studied by researchers. We conduct the first observational study through users' TAP debugging process with a 3-D smart home simulator and identify obstacles users face - users eventually fail to debug their TAPs in most cases when they encounter the obstacles. Furthermore, we develop two workflows that automatically fix trigger-action programs based on the misbehavior users experienced. The two workflows have been shown to help users overcome the obstacles and achieve better performance in TAP debugging. Our work also shows that it is feasible to study users' trigger-action programming and debugging experience in a 3D simulator. This reduces the overhead of time, cost, and privacy concerns of conducting TAP studies in the real world.

We hope that the mental obstacles summarized from our user study, the debugging workflows that we developed, and the TAP-enabled 3D simulator can all help future research on TAP programming and debugging.

ACKNOWLEDGMENTS

This material is based upon work supported by a gift from the CERES Center and by the National Science Foundation under Grants CCF-1837120 and OAC-1835890.

REFERENCES

[1] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. Scalable analysis of interaction threats in iot systems. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 272–285, 2020.
[2] Lorin W Anderson and Lauren A Sosniak. *Bloom's taxonomy*. Univ. Chicago Press Chicago, IL, USA, 1994.
[3] 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 *Proc. CHI*, 2019.
[4] Lei Bu, Wen Xiong, Chieh-Jan Mike Liang, Shi Han, Dongmei Zhang, Shan Lin, and Xuandong Li. Systematically ensuring the confidence of real-time home automation IoT systems. *ACM TCPS*, 2(3):22, 2018.
[5] Nico Castelli, Corinna Ogonowski, Timo Jakobi, Martin Stein, Gunnar Stevens, and Volker Wulf. What happened in my home? an end-user development approach for smart home data visualization. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 853–866, 2017.
[6] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. SOTERIA: Automated IoT safety and security analysis. In *Proc. USENIX ATC*, 2018.
[7] Ryan Chard, Kyle Chard, Jason Alt, Dilworth Y. Parkinson, Steve Tuecke, and Ian Foster. Ripple: Home automation for research data management. In *Proc. ICDCSW*, 2017.
[8] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. Cross-app interference threats in smart homes: Categorization, detection and handling. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 411–423. IEEE, 2020.
[9] Sven Coppers, Davy Vanacken, and Kris Luyten. Fortniot: Intelligible predictions to improve user understanding of smart home behavior. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(4):1–24, 2020.
[10] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. Empowering end users in debugging trigger-action rules. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2019.
[11] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. From users' intentions to if-then rules in the internet of things. *ACM Transactions on Information Systems (TOIS)*, 39(4):1–33, 2021.
[12] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. My iot puzzle: Debugging if-then rules through the jigsaw metaphor. In *International Symposium on End User Development*, pages 18–33. Springer, 2019.
[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proc. TACAS*, 2008.
[14] 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*, 2018.
[15] John D Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7(2):151–182, 1975.
[16] Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 19(1):1–28, 2012.
[17] Home Assistant. https://www.home-assistant.io/docs/automation/.
[18] Bing Huang, Hai Dong, and Athman Bouguettaya. Conflict detection in iot-based smart homes. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 303–313. IEEE, 2021.
[19] Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In *Proc. UbiComp*, 2015.
[20] Matthew Hughes. Mozilla's new Things Gateway is an open home for your smart devices. TheNextWeb, February 7, 2018.
[21] Hamada Ibrhim, Sherif Khattab, Khaled Elsayed, Amr Badr, and Emad Nabil. A formal methods-based rule verification framework for end-user programming in campus building automation systems. *Building and Environment*, 181:106983, 2020.
[22] Insider Intelligence. How IoT devices & smart home automation is entering our homes in 2020. Business Insider, Jan 6, 2020. https://www.businessinsider.com/iot-smart-home-automation.
[23] Keyu Jiang, Hanyi Zhang, Weiting Zhang, Liming Fang, Chunpeng Ge, Yuan Yuan, and Zhe Liu. Tapchain: A rule chain recognition model based on multiple features. *Security and Communication Networks*, 2021, 2021.
[24] Thorin Klosowski. Automation showdown: IFTTT vs Zapier vs Microsoft Flow. LifeHacker, June 26, 2016.
[25] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. Systematically debugging IoT control system correctness for building automation. In *Proc. BuildSys*, 2016.
[26] Marco Manca, Fabio Paternò, Carmen Santoro, and Luca Corcella. Supporting end-user debugging of trigger-action rules for IoT applications. *International Journal of Human-Computer Studies*, 123:56–69, 2019.
[27] Sarah Mennicken, David Kim, and Elaine May Huang. Integrating the smart home into the digital calendar. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2016.

[28] Walt Mossberg. SmartThings automates your house via sensors, app. Recode.net, 2014.
[29] Chandrakana Nandi and Michael D Ernst. Automatic trigger generation for rule-based smart homes. In *Proc. PLAS*, 2016.
[30] openHAB. https://www.openhab.org/.
[31] Mitali Palekar, Earlence Fernandez, and Franziska Roesner. Analysis of the susceptibility of smart home programming interfaces to end user error. In *Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW)*, 2019.
[32] Prolific. https://www.prolific.co/, 2022.
[33] Bernard Riera, F Emprin, D Annebicque, M Colas, and B Vigario. Home i/o: a virtual house for control and stem education from middle schools to universities. *IFAC-PapersOnLine*, 49(6):168–173, 2016.
[34] Vijay Srinivasan, Christian Koehler, and Hongxia Jin. Ruleselector: Selecting conditional action rules from user behavior patterns. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(1):1–34, 2018.
[35] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proc. CHI*, 2014.
[36] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.
[37] 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*, 2019.
[38] Ding Xiao, Qianyu Wang, Ming Cai, Zhaohui Zhu, and Weiming Zhao. A3id: an automatic and interpretable implicit interference detection method for smart home via knowledge graph. *IEEE Internet of Things Journal*, 7(3):2197–2211, 2019.
[39] Shaochun Xu and Václav Rajlich. Cognitive process during program debugging. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.*, pages 176–182. IEEE, 2004.
[40] Yinbo Yu and Jiajia Liu. Tapinspector: Safety and liveness verification of concurrent trigger-action iot systems. *arXiv preprint arXiv:2102.01468*, 2021.
[41] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. Autotap: Synthesizing and repairing trigger-action programs using ltl properties. In *Proceedings of the 41st International Conference on Software Engineering*, 2019.
[42] 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):1–26, 2020.
[43] 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*, pages 1–17, 2021.
[44] 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*, pages 1–10, 2020.

APPENDIX 1 - CODE BOOK

General Coding Criteria

Correctness

- 3: The modification was completely correct.
- 2: The modification solved the issues in the simulation, but either:
  - Introduced unrelated behaviors, or
  - Did not solve the issue completely if we changed the routine
- 1: The modification did not solve the problem, but was on the right track - e.g., wrong brightness level.
- 0: The modification was completely incorrect.

Issue Identification

Participant forgot about the task (Y/N).

- Y: Participants seemed to try to solve the wrong task (unrelated to the problem) - mark "Y" for participants who tried to solve more problems simultaneously.
- N: Participants were solving the correct task.
- Empty: This field can be left empty if coders are unsure. I assume this would not appear a lot.

Participants identified the wrong behavior (Y/N).

- Y: Participants identified the wrong behavior: they mentioned “something should have happened at an exact moment” or “this action shouldn’t have happened”. Note that in the Control group, if they solved the problem, coders need to select this option even if participants did not mention this.
- N: Participants did not identify the wrong behavior.
- Empty: This field can be left empty if participants did not specifically mention wrong behaviors. Please only leave it empty in cases of Explicit and Implicit interfaces.

Participants identified wrong behavior contexts (Y/N).

- Y: Participants identified contexts that enabled the wrong behavior - e.g. “I stood under the garage door so it did not close.”, or “I returned home before 11pm so the gate did not close.”. Note that in the Control group, if they solved the problem, coders need to select this option even if participants did not mention this.
- N: Participants did not identify wrong behavior contexts.
- Empty: Participants did not mention it.

TAP logic

Participants mis-understood if (event) vs while (state) (Y/N).

- Y: In the conversation, participants mentioned wrong understandings about “if” vs “while”. For example, “The if condition is constantly checking xxx”, or “when the while condition happens”.
- N: Participants understand it correctly in the conversation. Note that in the Control group, if they solved the problem, coders need to select this option even if participants did not mention this.
- Empty: Participants did not mention it.

Participants did not know "adding a condition" can prevent the action from happening (Y/N).

- Y: Participants mentioned something like “adding a condition would open it, but I would like to close it”, or “adding a condition will open it more”.
- N: Participants did not mention it.

Participants thought rules were dependent with each other (Y/N).

- Y: Participants thought that the rules were not independent, or needed a hint on that.
- N: Participants understood that rules were triggered independently, or did not mention it.

Participants tried to use unsupported syntax/semantics to solve the issue (Y/N).

- Y: participants tried to use branching/or condition/loops.
- N: participants did not try to use unsupported syntax/semantics.
- Empty: Explicit or Implicit.

Participants believed that a new rule would override the original rule (Y/N).

- Y: Participants mentioned that “this new rule will keep it off/closed/…”.
- N: Participants mentioned that the original rule would still happen.
- Empty: Participants did not mention it.

Participants understood behaviors of the selected/proposed modification (Y/N).

- Y: Participants successfully identified the behavior of a patch.
- N: Participants didn’t successfully answer the question.
- Empty: we did not ask.

Sensor understanding

Participants confused about levels (Y/N).

- Y: Participants said they are not sure about the (brightness) level, or specified the wrong level.
- N: Participants did not mention it; participants specified the correct level; or it was not applicable in the task.

Participants had a good understanding about related devices (Y/N).

- Y: In the follow up questions, participants identified correct related devices successfully.- We still mark it "Y" if participants mentioned correct things about sensors unrelated to the problem.
- N: They didn't identify related devices successfully.
- Empty: They didn't mention it (probably because we forgot to ask them the question).

Participants ignored relations between devices and sensors (Y/N).

- Y: Participants ignored the relation between lights and indoor brightness. Examples are shown below. In "nf" feedback interface, participants said "In the daytime it's level 5" on the indoor brightness, but actually level 5 was caused by lights.
- N: Participants figured out there was a relationship.
- Empty: It was N/A for the task, or participants did not mention it.

Attention

Participants received hints on reading (Y/N).

- Y: We mentioned to participants e.g., "this is about a different sensor", or "this is below, not above".
- N: We did not give hints about reading.

Participants misunderstood "go above/below level" statements (Y/N).

- Y: Participants misunderstood the statement. e.g., "Go above 0" means it's really dark
- N: Participants did not misunderstand. Note: to qualify for this option, participants needed to mention that explicitly.
- Empty: It was N/A for the task, or participants did not mention it.

Participants mis-read statements (Y/N).

- Y: Participants mis-read levels, above/below, outside/office, etc.
- N: Participants did not mis-read.

Interface specific

Participants received clarification about the interface (Y/N).

- Y: We gave additional clarifications about the interface. Some examples are shown below:
  - In Control, we told participants how to write the statement they wanted - "clock is in MISC", "to close the door, you need to select the F-Garage Door", etc.
  - In Explicit or Implicit, we clarified the meaning of the 4 selections - "the first two are about opening, and the last two are about closing", etc.
  - In Explicit, we clarified things in the feedback interface - "you only need to tell the system when the event should have happened.", "the change is not reflected in the later trace, but you don't have to worry about that.", etc.
- N: We did not give clarifications about the interface.

In which attempt did participants select the over-vs-under options right (1st, 2nd, etc.)

- How many attempts did participants take to select the correct statement among the "I would like the action xxx to xxx"s.
- If participants had not selected the right one at the end, put 0 here.

Stats and visualization helped participants make a decision (Y/N).

- Y: Participants used the information in the visualization.
- N: Participants did not use the information in the visualization.
- Empty: Control group.

The correct patch showed up in the suggested modifications (Y/N).

- Y: The correct patch showed up among the suggestions.
- N: The correct patch did not show up. Note: if we only ask them to go through the top 20, please select this option if it was not in the top 20.
- Empty: Control group.

Task-specific information

Task 1

- Completely correct (3):
  - Adding one of the following conditions to the "IF motion, THEN on" rule
    - Office brightness is below [1-2]
    - Outside brightness is below [2-7]
  - Duplicating the rule "IF motion, THEN on", each of them have the following conditions
    - Time is after [4pm-8pm]
    - Time is before [3am-7am]
- Partially correct (2):
  - Adding one of the following conditions to the "IF motion, THEN on" rule
    - Time is after [4pm-8pm]
- Incorrect, but good idea (1):
  - Adding the following conditions to the "IF motion, THEN on" rule
    - Office brightness is below [0, or 3-10]
    - Outside brightness is below [0-1, or 8-10]
  - Adding both of the following conditions to the "IF motion, THEN on" rule:
    - Time is after [4pm-8pm]
    - Time is before [3am-7am]
- Incorrect (0):
  - Adding a new rule about "close"
  - …

Task 2

- Completely correct (3):
  - Adding one of the following conditions to the "IF falls below 1, THEN open" rule
    - Outside brightness is above [0-7]
    - Bedroom brightness is below [2-5] - weird, but in the current implementation this could tell whether the "going below" was from turning off the lights
  - Adding both conditions to the "IF falls below 1, THEN open" rule
    - Time is before [12:30pm-10pm]

    - Time is after [5am-9am]
- Partially correct (2):
  - Adding one of the following conditions to the "IF falls below 1, THEN open" rule
    - Time is before [12:30pm-10pm] - did not cover cases when going to sleep after 12am
  - Adding both conditions to the "IF falls below 1, THEN open" rule - this has the issue that if we go to sleep before the time specified in "time before", the issue is not covered.
    - Time is before [10:01pm-11:59pm]
    - Time is after [5am-9am]
- Incorrect, but good idea (1):
  - Adding one of the following conditions to the "IF falls below 1, THEN open" rule
    - Time is after [7am-1pm] - did not cover the night time at all
- Incorrect (0):
  - Add a rule about "IF clock turns xxx . . . , THEN close the roller shade"
  - Add a rule about "IF brightness falls below xxx . . . , THEN close the roller shade"
  - Change the "IF falls below 1, THEN open" into "IF brightness falls below xxx . . . , THEN close"
  - Change the "IF goes above below 1, THEN open" into "IF brightness goes above . . . , THEN open"
    - This seems "partially correct" because it dismissed the wrong behavior. However, it introduced conflicted actions to the rules and might completely make the system not work
  - ...

Task 3

For this one, some participants in Control use "lights 1", but we consider it okay.

- Completely correct (3):
  - Change the time "18:20" into "[14:00-18:00]"
  - Add a new rule "IF clock turns [14:00-18:00], THEN on" with one or more of the following conditions:
    - None
    - Outside brightness is below [3-10]
    - Office brightness is below [1-10]
    - Motion is detected
    - Office/Outside Temperature is below [15-35] C - for distinguishing between winter/summer
    - . . .
  - Add a new rule "IF outside brightness falls below [3-10], THEN on"
    - It could have a condition about office brightness sensor
  - Add a new rule "IF motion, WHILE. . . , THEN on" with one or more of the 2 conditions:
    - Outside brightness is below [2-7]
    - Office brightness is below [1-2]
- Partially correct (2):
  - Add a new rule "IF office brightness falls below [1-2], THEN on"
  - It solved the issue. But it also prevented the light from turning off.
- Incorrect, but good idea (1):
  - Change the rule into "IF office brightness falls below level [3-10], THEN on"
- Incorrect (0):
  - Add a new rule "IF office brightness goes above xxx,... , THEN on"
  - Add a new condition to the original rule "IF 18:20, THEN on"
  - . . .

Task 4

- Completely correct (3):
  - Add a new rule "IF infrared stops, …, THEN close" with one of the following conditions:
    - None
    - Motion detector does not detects motion
- Partially correct (2):
  - Add a new rule "IF infrared stops, …, THEN close". On top of the correct conditions, it also has the following conditions:
    - Outside brightness is below [1-10]
    - Change "2 mins" to "[11-20] minutes".
- Incorrect, but good idea (1):
  - Add a new rule "IF motion stops WHILE infrared does not detect something, THEN close garage door"
  - Change "2 mins" to "[5-10] minutes".
  - Delete the condition "infrared does not detect something" in the original rule.
  - It solves the problem, but in a really dangerous way.
- Incorrect (0):
  - Adding any new condition to the the original rule
  - Add a new rule "IF infrared starts, …, THEN close"
  - Add a new rule "IF garage brightness falls below [1-10], WHILE…, THEN close"
  - Delete the "2 minutes", i.e. "IF garage door opens, …, THEN close".
  - Add a new rule "IF motion stops for xxx, THEN close"
  - …

Task 5

- Completely correct (3):
  - Add a new rule "IF clock becomes [23:00-23:30], THEN close" with none or any of the following conditions:
    - Outside brightness is below [1-10]
    - Motion detector does not detect motion
    - Entrance infrared [1,2] does not detect something
    - Alarm keypad is armed
    - Gate is open
  - Add a new rule "IF clocks become later than [23:00-23:30], THEN close" with none or any of the following conditions:
    - Outside brightness is below [1-10]
    - Motion detector does not detect motion
    - Entrance infrared [1,2] does not detect something
    - Alarm keypad is armed
    - Gate is open
    - Add a new rule "IF alarm keypad becomes armed WHILE gate is open, THEN close gate"
- Partially correct (2):
  - Add a new rule "IF [22:30-22:59]" with none or any of the following conditions - this would leave a gap, but it solves the problem in the simulation
    - Outside brightness is below [1-10]
    - Motion detector does not detect motion
    - Entrance infrared [1,2] does not detect something
    - Alarm keypad is disarmed

    * Gate is open
  - Add a new rule “IF infrared[1,2] stops detecting something, …, THEN close” with none or any of the following conditions - this solves the problem, but keeps the gate closed before 11
    * Outside brightness is below [1-10]
    * Motion detector does not detect motion
    * Alarm keypad is disarmed
    * Gate is open
    * Change the time “23:00” in “if clock turns 23:00, THEN arm” into “[18:00-22:50]”
- Incorrect, but good idea (1):
  - Add a new rule “IF infrared[1,2] starts detecting something, …, THEN close” with none or any of the following conditions - this solves the problem, but keeps the gate closed before 11. It’s also kind of dangerous because it closes the gate at the exact moment of coming in.
    * Outside brightness is below [1-10]
    * Motion detector does not detect motion
    * Alarm keypad is disarmed
  - Change the first rule “if gate opens,...” into any good rule specified in “correct fix”.
    * For example, change it into “if clock turns 23:00 while eg is open, then close eg”
- Incorrect (0):
  - Add a new rule “IF [08:00-xxx], …, THEN close”
  - Make any changes to the two alarm keypad rules
    * For example, add an action “close the gate” as a condition
  - Add the following condition to the “close entrance gate” rule:
    * The time is 23:00
    * Add a new rule “if brightness becomes xxx, THEN close”

APPENDIX 2 - RESULT ANALYSIS

Control

		Context	Action	Task	Logic	Dependency	Syntax	Level	Device	Relationship	Misreading	Behavior	Clarification
Task 1	Correct	0	0	0	0	0	0	1	0	0	0	0	0
	Incorrect	1	1	1	5	3	0	4	2	1	0	2	5
Task 3	Correct	0	0	0	0	0	0	0	0	0	0	0	0
	Incorrect	1	1	0	3	3	0	1	4	0	2	4	8
Task 4	Correct	0	0	0	0	0	1	1	0	0	0	0	5
	Incorrect	1	1	1	2	1	1	2	1	2	0	0	4
Task 6	Correct	0	0	0	0	0	0	0	0	0	0	0	0
	Incorrect	2	2	1	6	0	7	0	2	0	0	5	10
Task 12	Correct	0	0	0	0	0	0	0	0	0	0	0	2
	Incorrect	0	4	0	5	0	4	0	5	0	1	3	6
Overall	Correct	0	0	0	0	0	1	2	0	0	0	0	7
	Incorrect	5	9	3	21	7	12	7	14	3	3	14	33

Explicit-Feedback

		Context	Action	Task	Logic	Dependency	Syntax	Level	Device	Relationship	Misreading	Behavior	Clarification
Task 1	Correct	0	0	0	0	0	0	5	0	4	4	0	9
	Incorrect	0	0	0	1	1	0	0	0	0	1	1	1
Task 2	Correct	0	0	1	1	0	0	2	0	1	3	2	3
	Incorrect	0	0	1	5	3	0	0	1	0	3	2	5
Task 3	Correct	0	0	0	0	0	1	3	0	2	3	1	8
	Incorrect	0	0	0	1	0	0	2	1	1	2	0	2
Task 4	Correct	0	0	1	2	2	0	1	2	0	2	0	7
	Incorrect	1	2	0	1	1	0	1	2	0	1	0	3
Task 5	Correct	0	0	1	1	2	0	0	4	1	2	1	6
	Incorrect	0	2	1	1	0	0	0	3	0	2	0	4
Overall	Correct	0	0	3	4	4	1	11	6	8	14	4	33
	Incorrect	1	4	2	9	5	0	3	7	1	9	3	15

Implicit-Feedback

		Context	Action	Task	Logic	Dependency	Syntax	Level	Device	Relationship	Misreading	Behavior	Clarification
Task 1	Correct	0	0	0	1	1	0	2	0	0	4	0	2
	Incorrect	2	2	2	0	1	0	1	1	0	1	0	0
Task 2	Correct	0	0	0	0	0	0	1	0	0	3	1	2
	Incorrect	0	2	0	2	1	0	2	1	0	2	2	0
Task 3	Correct	0	0	0	0	0	0	1	0	0	5	1	1
	Incorrect	0	0	0	1	0	0	1	0	1	1	2	1
Task 4	Correct	0	0	0	0	2	0	1	0	0	0	1	2
	Incorrect	1	1	1	2	1	0	1	4	0	3	3	1
Task 5	Correct	0	0	0	0	0	0	0	0	0	0	0	0
	Incorrect	2	2	2	2	1	0	2	7	0	1	4	9
Overall	Correct	0	0	0	1	3	0	5	0	0	12	3	7
	Incorrect	5	7	5	7	4	0	7	13	1	8	11	11

APPENDIX 3 - TASKS

ID	Initial Goal	Problem	Error type	Discovered immediately	Estimated difficulty	Ideal modification
0	Make sure the garage door is never left open.	The garage door closes when people are under it.	Over-Automation	Yes	Medium	Add a WHILE condition
1	Turn on the light when people are present.	The light turns on even when it is bright.	Over-Automation	Yes	Easy	Add a WHILE condition
2	Allow more sunlight in the bedroom.	The blind opens when it's dark outside.	Over-Automation	Yes	Medium	Add a WHILE condition
3	Turn on the light when it gets dark.	The light turns on too late in winter.	Under-Automation	Yes	Easy	Change a parameter
4	Make sure the garage door is never left open.	The garage door is left open sometimes.	Under-Automation	No	Hard	Add a new rule
5	Keep the entrance gate closed during nighttime.	The entrance gate is sometimes left open throughout nighttime.	Under-Automation	No	Hard	Add a new rule