Trace2TAP: Synthesizing Trigger-Action Programs from Traces of Behavior

LEFAN ZHANG, University of Chicago
WEIJIA HE, University of Chicago
OLIVIA MORKVED, University of Chicago
VALERIE ZHAO, University of Chicago
MICHAEL L. LITTMAN, Brown University
SHAN LU, University of Chicago
BLASE UR, University of Chicago

Two common approaches for automating IoT smart spaces are having users write rules using trigger-action programming (TAP) or training machine learning models based on observed actions. In this paper, we unite these approaches. We introduce and evaluate Trace2TAP, a novel method for automatically synthesizing TAP rules from traces (time-stamped logs of sensor readings and manual actuations of devices). We present a novel algorithm that uses symbolic reasoning and SAT-solving to synthesize TAP rules from traces. Compared to prior approaches, our algorithm synthesizes generalizable rules more comprehensively and fully handles nuances like out-of-order events. Trace2TAP also iteratively proposes modified TAP rules when users manually revert automations. We implemented our approach on Samsung SmartThings. Through formative deployments in ten offices, we developed a clustering/ranking system and visualization interface to intelligibly present the synthesized rules to users. We evaluated Trace2TAP through a field study in seven additional offices. Participants frequently selected rules ranked highly by our clustering/ranking system. Participants varied in their automation priorities, and they sometimes chose rules that would seem less desirable by traditional metrics like precision and recall. Trace2TAP supports these differing priorities by comprehensively synthesizing TAP rules and bringing humans into the loop during automation.

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

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

ACM Reference Format:
Lefan Zhang, Weijia He, Olivia Morkved, Valerie Zhao, Michael L. Littman, Shan Lu, and Blase Ur. 2020. Trace2TAP: Synthesizing Trigger-Action Programs from Traces of Behavior. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3, Article 104 (September 2020), 26 pages. https://doi.org/10.1145/3411838

1 INTRODUCTION

Consumer Internet of Things (IoT) smart devices have become very common [28]. With automation, these smart devices can react to environmental contexts and user behaviors, potentially improving users' quality of life by

Authors' addresses: Lefan Zhang, University of Chicago, lefanz@uchicago.edu; Weijia He, University of Chicago, hewj@uchicago.edu; Olivia Morkved, University of Chicago, oliviamorkved@uchicago.edu; Valerie Zhao, University of Chicago, vzhao@uchicago.edu; Michael L. Littman, Brown University, mlittman@cs.brown.edu; Shan Lu, University of Chicago, shanlu@uchicago.edu; Blase Ur, University of Chicago, blase@uchicago.edu.

2474-9567/2020/9-ART104
https://doi.org/10.1145/3411838

streamlining their daily routines [1, 53]. However, automation is only helpful if it aligns with the user's intent. How to efficiently translate user intent into desired automation remains a crucial and challenging open problem.

Past attempts to address this problem mainly follow two directions. The first direction relies completely on users to program their smart devices. For example, through **trigger-action programming (TAP)** [17, 23, 56, 57], users write if-this-then-that **rules** (e.g., "IF *trigger* occurs WHILE *conditions* are true, THEN take some *action*"). The second direction relies completely on automated prediction leveraging statistical or machine learning techniques. That is, automated analysis of users' past behaviors produces a model that predicts and automates future interaction between users and devices. Both directions have limitations.

The first direction, having users write rules, often excels at simple scenarios. Unfortunately, in the nuanced and complex device-automation scenarios that frequently arise in homes or other environments with more than a handful of devices, users may write rules with bugs [5, 23, 38] or struggle to understand why particular automations are running [39, 40, 61]. In short, while prior work has shown that even non-technical users can write simple trigger-action programs with ease [56], users struggle to communicate their intent via rules when the intent they wish to communicate becomes more complicated, involves particular devices being controlled simultaneously by multiple rules, or involves automation actions that trigger based on opaque sensor readings.

The second direction, building an automated predictor using statistical or ML techniques [3, 51], excels at finding a model or a program that best emulates past user behaviors based on metrics like precision and recall. Unfortunately, maximizing precision and recall based on past behaviors does not necessarily best capture an individual user's intentions, priorities, and concerns. Notably, what type of automation is "best" varies across users. Some may want to automate a particular context more than others, while others may care more about precision or rule generalizability. Furthermore, while a statistical or ML algorithm is trained exclusively on past behaviors, a user's true intention may not be clear from past behaviors. In a home or office environment filled with sensors, there will be many spurious correlations between sensors and actions that should not be automated. On the other hand, the intended automation may not even exist verbatim among the observed behaviors. For example, a user may want the light to turn off right *after* they leave the kitchen, yet the location of the light switch means that the automated system always observes them turning the light off moments before leaving the kitchen. As another example, the trace may show that a light was often kept on for the whole night, but that may not reflect the user's intent—the user may happen to be a forgetful person. Finally, it is exceedingly difficult to bring a non-technical human into the loop with most ML classifiers. For many types of automated predictors, it is difficult both to explain to the user what exactly the predictor does and to enable the user to edit the predictor.

To better support device automation, in this paper we propose and evaluate a new hybrid approach that combines the respective strengths of trigger-action programming and automated learning. Our approach, **Trace2TAP**, takes as input a trace of user behavior, or time-stamped log of all sensor/environmental readings and events, as well as all instances of humans manually taking actions (e.g., turning on the air conditioning by pressing the "on" button in a smartphone app in real time). Trace2TAP automatically synthesizes TAP rules that could automate a good portion of the observed instances of human actions in a *comprehensive* way to accommodate for users' diverse priorities and concerns. To align the user's intent with what is being automated, Trace2TAP presents the user with the synthesized rules and visualizes for them the rationale for each rule based on the trace. Because there are often myriad synthesized rules, including variants with subtle differences, Trace2TAP clusters rules based on the similarity of their actuations and ranks both clusters and the rules within each cluster based on a number of relevant characteristics. Furthermore, Trace2TAP aids in debugging by suggesting **patches** (modified rules or new rules) based on observations of automations being reverted (e.g., a human closing the shades immediately after a rule had automatically opened them).

Our Contributions in Designing and Evaluating Trace2TAP

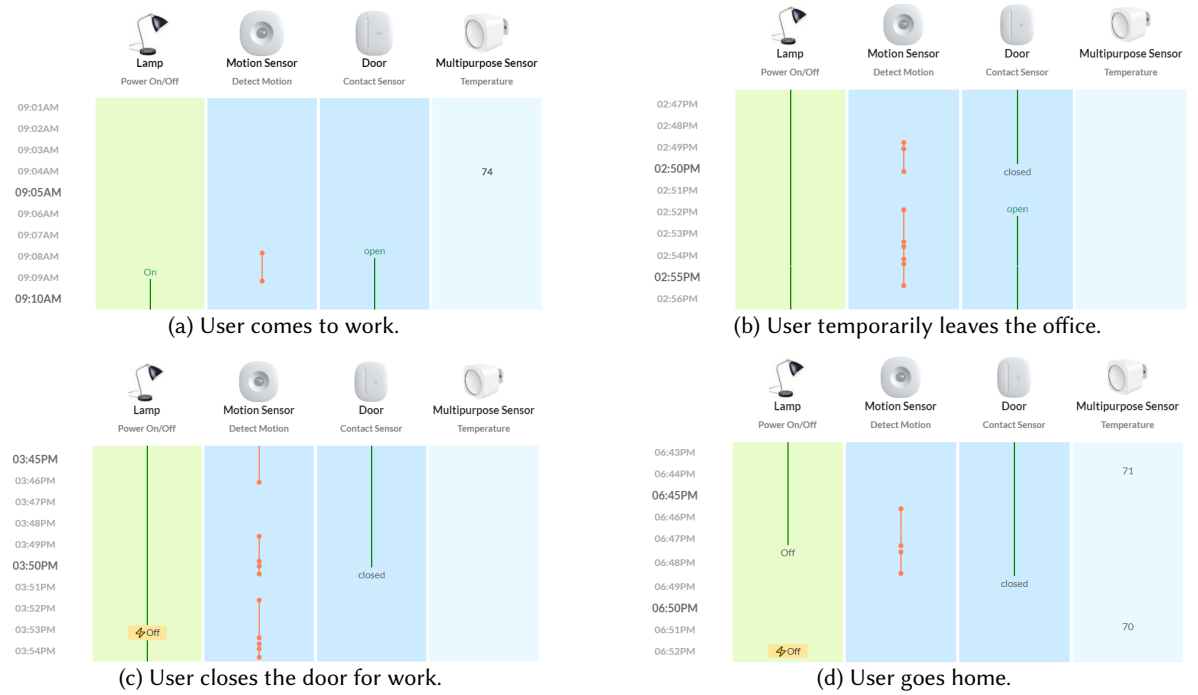
Clearly, Trace2TAP cannot simply combine existing techniques to achieve the aforementioned vision. As detailed below, we developed a novel synthesis algorithm to comprehensively generate TAP rules, designed a new clustering/ranking scheme to help end users navigate among all the synthesized rules, designed user interfaces for explaining TAP rules based on the traces from which they were synthesized, and fully implemented our approach on top of Samsung's SmartThings platform. We also conducted a user study applying our system to the control of smart devices in a workplace environment, encompassing formative field deployments in ten offices to aid in the development of Trace2TAP, followed by a summative evaluation field study in seven offices.

1) A novel algorithm for rule synthesis: For every device action to automate (e.g., "turn on the light"), Trace2TAP first identifies a set of device capabilities (termed **variables**) that are statistically correlated with the action (Section 5.1). It then applies symbolic execution and SAT-solving techniques [11] to exhaustively generate all possible rules involving those variables that can automate more than a threshold portion of instances in the trace of the user manually taking that action (Section 5.2). Our novel use of symbolic reasoning and SAT-solving techniques for synthesizing TAP rules enables Trace2TAP to be both *automated* (synthesizing rules automatically from observed behaviors) and *comprehensive* (producing a large number of rules that can approximate users' past device interactions in various ways). The latter property captures the spectrum of users' priorities and intentions.

2) A novel prioritization and visualization scheme for rule presentation: A key goal was for Trace2TAP to bring the human into the loop intelligibly and efficiently. To help users avoid redundancy and choose the rules that best capture their intent, Trace2TAP clusters the many rules synthesized to automate an action (Section 6.1). Clusters represent rules that automate similar instances of the user manually performing that action, as captured in the trace. Within each cluster, Trace2TAP ranks the rules based on six features we identified through formative deployments in ten offices. To help users understand the relationship between a synthesized rule's potential automations and the manual instances of those actions from the trace, Trace2TAP visualizes the state of the various sensors at points in the trace the synthesized rule would have triggered (Section 6.2).

3) A mixed-methods empirical field study: We validated Trace2TAP through a summative field study in seven offices. We installed commercial sensors and smart devices in each office. Participants manually controlled the devices as they normally would. After four months of usage, we conducted a semi-structured interview during which participants used Trace2TAP to choose their automations, enabling us to gauge the alignment between the participant's intent and the rules Trace2TAP synthesized. Trace2TAP successfully generated TAP rules that participants selected to automate almost all manual actions in their offices. We found that participants selected rules that were frequently ranked highly by our clustering and ranking approach; the median rank of selected rules was second. We also found our clustering scheme to be effective; participants almost never selected more than one rule from a given cluster. Participants sometimes chose rules that would seem less desirable based only on quantitative metrics, highlighting the value in Trace2TAP comprehensively generating a variety of TAP rules and making those proposed automations intelligible to users, who could then make informed decisions. Furthermore, this field study uncovered a number of lessons for why participants might reject certain rules that otherwise seem promising. It also highlighted subjective factors influencing participants' approach to automation, as well as best practices for sensor placement and sensor visualization.

4) A novel debugging approach: After a user chooses a set of TAP rules, Trace2TAP continues to help align device automation with the user's intent by observing and acting upon cases when the user implicitly demonstrates dissatisfaction with an automation. When a user manually reverts an automation caused by a rule, Trace2TAP automatically synthesizes rule patches, or proposed modifications to the set of rules, using a similar symbolic reasoning and constraint-solving framework (Section 5.3). The rule-presentation interface mentioned above then helps the user understand the impact of each patch and make a proper debugging decision.



(a) User comes to work.

(b) User temporarily leaves the office.

(c) User closes the door for work.

(d) User goes home.

Fig. 1. Visualizations of different use cases in the trace collected from an example office occupant.

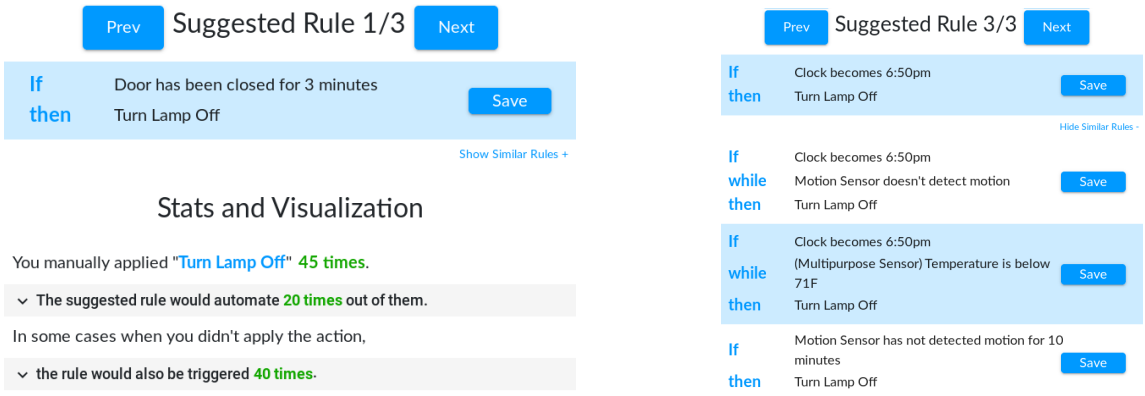
We have open-sourced Trace2TAP,[1] including implementations of the algorithms and our software integration with Samsung SmartThings. We are also releasing our user study materials and, with participants' opt-in permission, the raw traces collected in our field study. We encourage reuse of both our data and Trace2TAP tool.

2 END-TO-END EXAMPLE OF TRACE2TAP AND ITS USER EXPERIENCE

Before detailing Trace2TAP's algorithms, implementation, and evaluation in subsequent sections, here we provide an example of Trace2TAP's end-to-end usage. This example describes the automations Trace2TAP suggested for controlling the lights in one of the ten offices in our formative initial deployment. In doing so, it also distills key elements of our conceptual approach in designing Trace2TAP's algorithms and user experience.

Among the devices the occupant chose for their office were Philips Hue lights, which they used as primary illumination. At a high level, the occupant's intent was for the lights to be on while they were in the office and off otherwise, with a weak preference for the lights to remain on as a signaling mechanism that they would return soon when they had momentarily stepped out of their office. For two weeks, the occupant of this particular office manually operated the lights with a wireless Philips Hue light switch located on their desk. Sensors recorded motion in two parts of the office, the office's temperature and illumination (via a multipurpose sensor), whether the door was open (via a contact sensor), and the time of each reading. As observed in the trace collected (see Figure 1), the occupant typically turned the lights on upon arrival in the morning, though their arrival time was variable. When they left their office for extended periods of time or at the end of the day, they turned their lights off. They occasionally left their office for short restroom breaks, leaving the lights on during them. While in their office, they usually left the door open. However, they also sometimes closed their door to concentrate.

[1]The Trace2TAP algorithm and code, integration with SmartThings, and evaluation data are available at https://github.com/zlfben/trace2tap



(a) Rule 1 in Cluster 1 for turning off the lamp.

(b) All rules in Cluster 3, shown on demand.

Fig. 2. An example of Trace2TAP's UI for showing proposed rules to the occupant.

Trace2TAP synthesized and proposed numerous rules for turning the lights on and off, yet clustered these rules into just a few clusters. Note that this particular office contained multiple lights controlled identically, and Trace2TAP's rules for the lamp were the same for all other fixtures. For turning the lamp on, the rule Trace2TAP ranked highest (first and only cluster, first rule) was straightforward: *'IF door opens THEN turn on the lamp.'* While this rule would try to automate turning the lamp on many times when the lamp was already on, Trace2TAP intentionally only labels automations as false positives if they would put the device in a state different from that observed in the trace. This rule was also ranked higher than many other rules because it is compact, not involving any additional conditions (e.g., as opposed to *'IF door opens WHILE the temperature is 71°…'*). While more granular rules might fit the trace's data more precisely, more compact rules tend to generalize better. Trace2TAP presented the proposed rules to the occupant through a UI analogous to Figure 2. The UI also visualized relevant sensors and devices from the trace, as in Figure 1. Upon demand, the user could show all rules in the cluster, as in Figure 2b. We developed the ranking system, UI, and visualizations iteratively during our formative deployment.

The three clusters of rules synthesized for turning the lights off were more nuanced than for turning them on. Proposed rules for turning them off needed to account for the occupant briefly leaving the office. Trace2TAP proposed *'IF door has been closed for 3 minutes THEN turn the lamp off,'* which Trace2TAP ranked highly and also initially seemed to match the occupant's intent. The occupant looked through the different clusters before choosing this rule, sometimes looking at the additional rules in the cluster.

Another strength of Trace2TAP relative to prior work is that, even after TAP rules have been manually written by the user or synthesized by the tool, Trace2TAP continues to use the trace to propose patches (modifications). Instances of the user reverting rules' automations or manually controlling devices beyond the current automations both contribute to patches. As such, Trace2TAP provides human-intelligible proposals for automation throughout a workflow that simultaneously supports manual trigger-action programming for the communication of intent.

This advantage was critical in adjusting the rules. Even though the rule ranked first in Cluster 1 for turning off the light (Figure 2) captured most use cases, it turned the light off when the occupant closed their door to concentrate. In subsequent days, the occupant turned the light back on multiple times when that automation occurred, so Trace2TAP proposed a patch (Figure 3) that added an additional condition: "Motion Sensor doesn't detect motion." At a high level, while the door being closed for more than a few minutes was normally correlated with the lights being turned off, the continuing trace also showed cases in which the occupant stayed in their office when the door was closed, registering motion on the motion sensors and wanting the lights to be on.

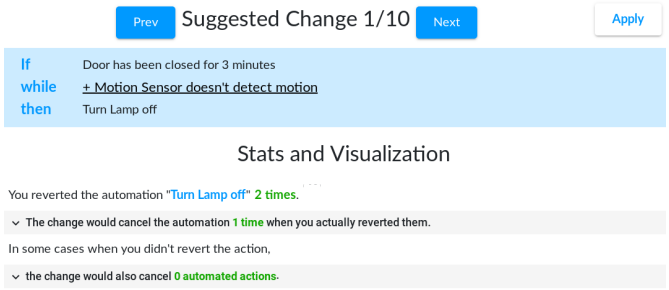


Fig. 3. The patch Trace2TAP suggested to modify Rule 1 from Cluster 1 based on the user reverting automations.

Multiple aspects of Trace2TAP contributed to its ability to identify and prioritize an appropriate rule, and then to refine it. Unlike most prior work (Section 3), Trace2TAP's algorithm handles events that appear out of order. Because of the physical location of the wireless light switch in the office, the occupant first turned off the lights and *then* closed the door. Nonetheless, both the temporal aggregation phase of Trace2TAP's pre-processing and its ability to create time-based triggers (Section 5) account for out-of-order events in the trace. Furthermore, Trace2TAP's comprehensive approach to TAP rule synthesis across variables enabled it to identify an appropriate rule combining sensors in complex ways, presenting the most promising possibilities to the user in an intelligible way to enable them to filter the rules through their own intent.

3 RELATED WORK AND BACKGROUND

We first highlight how our work fits into the broader literature on context-aware computing and trigger-action programming. We then compare Trace2TAP with prior systems for synthesizing automations in smart spaces and provide background on the computational techniques that serve as Trace2TAP's foundation.

3.1 Trigger Action Programming (TAP) in Smart Spaces

User-written TAP rules have been used as a programming interface for controlling physical smart devices in widely used [41, 57] systems including IFTTT [27], Zapier [33], Microsoft Flow [35], Samsung SmartThings [44], Mozilla Things Gateway [25], OpenHAB [48], and Home Assistant [22]. Initial user studies found that non-technical users could accurately write TAP rules to automate smart spaces in simple scenarios [17, 45, 56, 59]. However, more recent work has observed shortcomings of TAP in more complex scenarios [6, 9], in particular relating to users writing rules that contain bugs or otherwise fail to match their intent [5, 23, 61]. Furthermore, users often find it hard to reason about how sensors (e.g., motion sensors) in smart homes work [21]. For example, they may misunderstand what brightness-level readings mean.

Tools have been developed to detect some bugs in TAP programs [10, 13, 55, 58]. However, it is difficult in the general case to discover all bugs. Furthermore, even with a bug detector, developers still need to manually write rule patches, which is challenging. Recently, several tools have been built to not only detect, but also automatically fix, bugs in TAP programs. Some focus on specific types of bugs, like missing-trigger bugs [46]. Others leverage constraint solving and/or model checking techniques to automatically synthesize TAP rules or rule patches that satisfy specific constraints [7, 36, 62]. Although powerful, the capabilities of these synthesis tools are limited to the correctness properties made available to them (e.g., 'windows opening' and 'raining' should never occur simultaneously). Eliciting properties or requirements from users often requires them to map their intent to those systems' input templates, which may still be difficult for users. This may even be impossible for some user intents due to the limitation of those systems' property-input templates. In contrast to this prior work, Trace2TAP follows a complementary approach of taking traces as input and then modifying TAP programs based on observations of instances that are not yet automated and observations of the user reverting automations.

3.2 Automating Smart Spaces From Traces

While Trace2TAP is the first system to comprehensively synthesize TAP rules from traces as part of a human-in-the-loop framework, a number of prior systems have used other algorithms and other user experience approaches to predict user activities and/or automate smart devices based on traces. Below, we briefly describe the different features of each approach and comment on their common limitations.

The CASAS system analyzes a smart home's trace and utilizes machine learning techniques to generate automation policies [51]. The automation policy itself is completely hidden from users. Users can only indirectly refine the automation policy by expressing whether they like, or do not like, some of the automation sequences. Minor et al. take a trace as input and use imitation learning algorithms to predict future human activities [42, 43]. The prediction model is completely based on the trace. It is not meant to be intelligible to, or adjustable by, human users. Furthermore, the model focuses on predicting high-level activities like "having dinner" instead of directly automating specific devices. The PUBS system [3] discovers frequent patterns from users' traces and turns them into event-driven rules that represent a subset of TAP rules. Similar to Trace2TAP, it enables users to see the synthesized automation in the form of TAP rules. In contrast, Trace2TAP uses symbolic constraint solving to generate rules more comprehensively, including rules that do not follow the exact event order in the trace.

Finally, a key difference that distinguishes Trace2TAP from all of these prior systems is that all the prior work treats the trace as a precise "role model," finding the automation that most closely mimics past behaviors observed. However, our field study (Section 9) highlights that there is often a gap between users' past behavior and their ideal automations. In contrast to prior work, Trace2TAP instead exhaustively synthesizes a comprehensive list of programs that can automate some portion of users' past behavior in a more generalized way. Such an approximate-search approach improves the odds of covering users' ideal automations. Moreover, Trace2TAP also clusters and visualizes the impacts of the rules it synthesizes to make the prospective rules intelligible for users.

3.3 Program Synthesis with Constraint Solving

To synthesize TAP rules, Trace2TAP turns its requirements about TAP rule candidates into symbolic constraints and leverages an existing constraint solver, Z3 [11], to exhaustively generate solutions. A constraint solver finds value assignments of a set of binary, numeric, or enumerate symbols that satisfy given constraints, a problem often referred to as a "Satisfiability Modulo Theories (SMT)" problem. Although even an SMT sub-problem, the Boolean satisfiability problem (SAT), is NP-complete [18], state-of-the-art SMT solvers deploy smart searching strategies [12] to solve typical SMT problems efficiently. They are widely used in static program analysis, hardware/software verification, and test-case generation [12].

SAT/SMT solvers are efficient tools to explore the search space in program synthesis [19]. Previous work has synthesized various types of programs using logical specifications [29, 54] or input-output examples [31] as input constraints. Once the input constraints are set, the program search space could be a hole in the program, several lines of loop-free code [31], regular expressions [47], or more [19]. Compared to prior SMT-based program synthesizers, Trace2TAP is unique in how it obtains and handles constraints, its search space, and its human-in-the-loop presentation of the results. It is unique in setting its requirements based on the device-interaction traces and does not require explicit input or specifications from users. It is unique in symbolically executing the trace to translate users' implicit requirements into constraints. To our knowledge, Trace2TAP's search space of trigger-action programs also differs from all prior work. Furthermore, Trace2TAP clusters and ranks the synthesized rules to enable intelligible and scalable presentation to users.

3.4 Context-Aware Computing

Context-aware systems customize services based on the context of usage [14]. By absorbing information about complex and dynamic real-world environments, systems can adapt to better meet a user's needs, such as

providing different recommendations based on the user's location or running apps in different modes based on the user's style. Previous work has enabled systems to automatically adapt their behaviors to optimize resource management [34, 50], provide personalized services [63], and even make crucial decisions regarding security and privacy [2, 32]. Unlike Trace2TAP, they do not aim to automate most of the manual interaction with devices.

Some work has argued that users need control in a context-aware system [4, 15, 16, 40, 60]. Instead of making decisions fully on the user's behalf, a context-aware system should communicate and collaborate with users. Recent work has thus attempted to provide useful information about the system to users [26, 30, 37] and designed interfaces that establish constructive collaboration between systems and users [8, 20, 39].

4 DEFINITIONS AND TERMINOLOGY

To make further discussion easier, we formally define a few important concepts below:

A **variable** represents a device capability. A smart home system with a color light and a thermostat would contain variables `light.switch`, `light.brightness`, `light.color`, `therm.current_temperature` and `therm.temperature_setting`. We collect all the variables for each device based on SmartThings [44] API manuals.

A **value** is the setting of a variable at a given time (e.g., a switch variable could be valued "on" or "off").

A **state** is the union of all variables' values in the system at a given time. It can be regarded as a dictionary with variables as keys. We will refer to a state as S : State, where $S[V]$ is the value of variable V in state S.

An **event** is what happens in the system that causes a state change, like "the light gets turned on" or "temperature drops below 0 degrees." We denote an event as E : Event, and $E.var$ is the variable whose value is changed to $E.val$ by E. For the event "the light gets turned on," $E.var$ is `light.switch` and $E.val$ is on. An event can be external or automated. External events are applied by the environment or manually by users, and automated events are triggered by the TAP automation system. An **action** is a type of event that can be sent to an actuator as a command, such as "turn on the lamp" or "mute the speaker."

A **trace** is the history of a smart device system over a time period. A trace T : Trace has 3 fields. T.init : State represents the system state at the beginning of the trace; T.events : list(Event) is a list of events that happened during the trace's time period; T.timestamps : list(Time) is a list of the timestamps of the events.

A **proposition** is a Boolean expression that compares the setting of a variable V with a constant value K, denoted as "V op K."

In Trigger-Action Programming (TAP), each TAP program consists of a set of TAP **rules**. In this paper, we use a variant of TAP rules with the format "IF **trigger** happens WHILE **conditions** are true THEN apply **action**" [5]. The **trigger** is an *event* and the **conditions** are a set of *propositions*. This type of TAP rule is deployed in some smart home frameworks [22, 44], and prior work has found that it is the least ambiguous TAP format [5, 49].

As we will explain in Section 5.2, to synthesize a TAP rule that can trigger a specific action, Trace2TAP essentially needs to synthesize a **trigger** proposition and **condition** propositions. For example, a trigger-proposition "`light.switch == on`" corresponds to "IF the light gets turned on" in a TAP rule. A condition-proposition "`therm.current_temperature > 70`" corresponds to "WHILE temperature is above 70 degrees" in a TAP rule.

5 TRACE2TAP RULE SYNTHESIS ALGORITHMS AND PROCEDURE

In this section, we present how Trace2TAP *automatically* synthesizes TAP rules in a *comprehensive* manner.

Inputs: Trace2TAP takes as input a trace T, the list of rules $R_1, ..., R_m$ already installed in the system, and an action $\mathbb{A}$ to automate (e.g., turn on the light).

Outputs: Trace2TAP will synthesize a list of TAP rules.

For each rule R to be synthesized, although its final presentation by Trace2TAP will have the intuitive form "IF **trigger** WHILE **conditions** THEN **action**," the raw output of this synthesis component includes one trigger

proposition, denoted as $V_t\ op_t\ K_t$, and one or multiple condition propositions, denoted as $V_c\ op_c\ K_c$. Note that the "action" component of R must be $\mathbb{A}$ and hence needs no further discussion.

For example, to synthesize a rule to automatically turn on the light, Trace2TAP may synthesize three components to form the trigger proposition: (1) `illuminance` for the variable V_t; (2) $<$ for the operator op_t; and (3) 100 lux for the constant value K_t. It may also synthesize three components to form the condition proposition: (1) `door.open` for the variable V_c; (2) == for the the operator op_c; and (3) `TRUE` for the constant K_c. This result will be post-processed and then presented as "IF the illuminance in the room drops below 100 lux WHILE the door is open THEN turn on the light."

Goals: Trace2TAP aims to synthesize many TAP rule candidates, as many as it can, that approximate over a threshold portion of action $\mathbb{A}$'s instances in the trace T. Here, approximating an action instance A means that if the rule R was in place, R would automatically trigger A close to the original moment when A took place. This goal aims to be *comprehensive* and *approximate*. First, we intentionally do not require the rule to trigger A exactly at its original moment as a user's intention may not be to exactly repeat what they did manually. For example, automatically turning off the light 30 seconds after the user typically had done so manually could be fine, or even more desired. Second, we intentionally do not require the rule to trigger all or most instances of the action $\mathbb{A}$ because a given action could occur under different contexts and hence may require more than one rule to automate. For example, one may turn off a light because the room is empty or because it is already bright outside.

Solution overview: A naive solution is to enumerate every possible TAP rule with $\mathbb{A}$ as its action and then check how many instances of action $\mathbb{A}$ in the trace could have been approximated by this rule. This clearly would consume too much time to be practical. In fact, such a rule enumeration might never finish because the constant values of the trigger and condition propositions, K_t and K_c, might take on an infinite number of settings.

Trace2TAP uses a novel two-step solution. It first identifies top *variable* candidates that are most suitable for the trigger and condition propositions, respectively, by applying signal processing techniques to analyze the trace T. It then formulates a symbolic constraint that represents what type of TAP rules, composed of those identified variables, can approximate over a threshold portion of action $\mathbb{A}$ instances in the trace. Consequently, instead of enumerating through all possible TAP rules, Trace2TAP simply feeds the symbolic constraint into a constraint solver, getting *all* the constraint-satisfying TAP rules generated.

The first step is crucial to avoid state-space explosion problems in the later constraint solving. It will be explained in Section 5.1. The second step is the key that allows Trace2TAP to be *inclusive*. It will be explained in Section 5.2. Finally, we will also discuss how a small adaptation to the Trace2TAP rule synthesis framework also allows Trace2TAP to automatically synthesize rule patches in Section 5.3.

5.1 Rule Synthesis: Variable Selection

We discuss below how to select top candidate variables that constitute **trigger** propositions (V_t) and **condition** propositions (V_c), respectively, based on their different roles in a TAP rule.

Finding trigger variables. Our first step is to identify a potential trigger for our TAP rule. By definition, a TAP rule conditionally causes an action to take place *right after* the triggering event occurs. Consequently, for a TAP rule to automate a good portion of instances of the action $\mathbb{A}$ near their original occurrences in the trace, its trigger variable should have a *temporal* relationship with $\mathbb{A}$. For example, if we want to automatically turn on the heater and the trace shows that the heater is often turned on shortly after the door is opened, the variable "`door.open`" can be a candidate to form the trigger of the TAP rule. On the other hand, if a certain capability of a device (i.e., a variable) rarely changes its setting within a short time window around the to-be-automated action $\mathbb{A}$, this device capability is unlikely to form the rule trigger we are looking for.

To evaluate whether variable V : Variable has a temporal relationship with action $\mathbb{A}$: Event, we regard all events related to V in the trace (i.e., all E so that $E.var == V$) and all instances of the action $\mathbb{A}$ as two sets of pulses, with

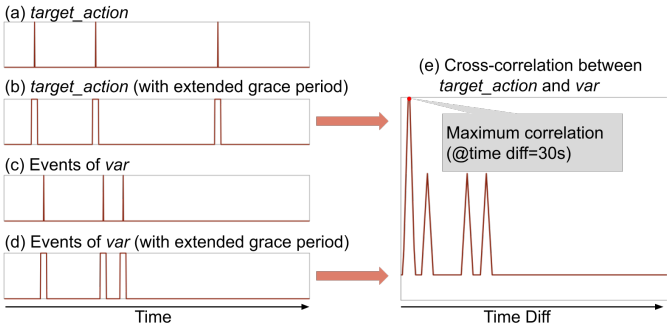


Fig. 4. Calculating how a variable *var* is related to *target_action*.

each pulse lasting for a certain time period (15 minutes by default in our current implementation), as illustrated in Figure 4. We then calculate the cross-correlation between these two signal sequences. Cross-correlation is used frequently in signal processing to measure the similarity between two signals based on sliding convolution.

To accommodate for the potential time gap between a trigger event and a to-be-automated action instance, both to make Trace2TAP inclusive and because rule triggers can contain time buffers (e.g., "10 minutes after Alice leaves the house"), we also evaluate the cross-correlation between the two signal sequences at various time differences, ranging from 1 second to 30 minutes. Trace2TAP then records the time difference $\texttt{time}_\Delta$ with the highest correlation for each variable V, ($\texttt{time}_\Delta$, $\texttt{correlation}_{max}$).

Finally, for all variables in the system, Trace2TAP picks the three variables[2] with the highest $\texttt{correlation}_{max}$ values. These three variables will be considered as candidate trigger variables, denoted as $V_{t_1}, V_{t_2}, V_{t_3}$. If a candidate variable uses a $\texttt{time}_\Delta$ greater than one second to achieve its highest correlation, we store its $\texttt{time}_\Delta$ with it, which can be used later to compose temporal TAP rules (Section 5.2.4).

Finding condition variables. Recall that our TAP rules take the form "IF trigger happens WHILE conditions are true THEN apply action." Each rule must have a single trigger, and that trigger must represent an event that occurs at a discrete moment in time (e.g., a door changes from the closed state to the open state). However, conditions have very different semantics [5], and condition variables serve a different role from trigger variables. A rule may have zero or more conditions, each representing a state that is true or false (a proposition). For instance, a condition might test whether a door is currently in the open state, and when it entered into that state is irrelevant. A condition variable V_c therefore does not need to correlate with the rule action $\mathbb{A}$ in a temporal way. The variable's value does not need to change around the time when the rule action is triggered; it just needs to stay at a particular value or value range around that time.

Consequently, we treat this as a conditional information problem. Each variable can be considered a random variable if we sample it throughout a trace, and we can calculate the entropy, a "randomness" measurement, of the variable based on the distribution of its value in the trace. If the variable is related to whether or not to apply a specific action $\mathbb{A}$, the distribution of its value will be less random if we sample its value when an instance of $\mathbb{A}$ occurs in the trace. To evaluate how much randomness we can reduce by sampling at instances of $\mathbb{A}$, we calculate the variable's conditional entropy at those times. The difference between the two entropies represents how predictable the variable becomes when $\mathbb{A}$ occurs. The more predictable it is, the more likely it can be used in

[2]Trace2TAP can be configured to pick more trigger or condition variables. We use three as our default setting because the complexity of the rule constraint solving grows exponentially with the number of variables and three variables was sufficient in our experience.

Table 1. A symbolic TAP rule. $\boldsymbol{\lambda}$'s, $\otimes$'s, $\boldsymbol{K}$'s, $\boldsymbol{\mu}$'s, $\oplus$'s are symbols; V_{t_*} and V_{c_*} are candidate variables for the rule's **trigger** and **conditions**, respectively.

$$\text{IF } \begin{cases} \boldsymbol{\lambda}_{t_1} \Rightarrow V_{t_1} \otimes_{t_1} \boldsymbol{K}_{t_1} \\ \boldsymbol{\lambda}_{t_2} \Rightarrow V_{t_2} \otimes_{t_2} \boldsymbol{K}_{t_2} \\ \boldsymbol{\lambda}_{t_3} \Rightarrow V_{t_3} \otimes_{t_3} \boldsymbol{K}_{t_3} \end{cases} \text{, WHILE } \begin{matrix} \boldsymbol{\mu}_{c_1} \rightarrow V_{c_1} \oplus_{c_1} \boldsymbol{K}_{c_1} \\ \boldsymbol{\mu}_{c_2} \rightarrow V_{c_2} \oplus_{c_2} \boldsymbol{K}_{c_2} \\ \boldsymbol{\mu}_{c_3} \rightarrow V_{c_3} \oplus_{c_3} \boldsymbol{K}_{c_3} \end{matrix} \text{, THEN take action } A.$$

Symbol	Name	Description
$\otimes, \oplus$	Comparator symbols	Each can be "=" (become), "≠" (change from), ">" (become greater than) or "<" (become smaller than), depending on the variable type.
$\boldsymbol{K}$	Value symbols	Each can be true or false for boolean variables, set options for set variables, or a number for range variables.
$\boldsymbol{\lambda}, \boldsymbol{\mu}$	Selection symbols	Each can be true or false, indicating whether its corresponding trigger or condition proposition is selected to form the rule. For example, if λ_{t_1} is true while λ_{t_2} and λ_{t_3} are false, the new rule's trigger is "$V_{t_1} \otimes_{t_1} K_{t_1}$".

a rule condition. We rank all variables by the following value:

$$\frac{\text{entropy(throughout trace)} - \text{entropy(at } \mathbb{A})}{\text{entropy(throughout trace)}}$$

The top three variables are selected as potential variables to constitute rule conditions, denoted as $V_{c_1}, V_{c_2}, V_{c_3}$.

5.2 Rule Synthesis: Symbolic Constraint Solving

Given the candidate variables, Trace2TAP takes three steps to synthesize trigger and condition propositions: (1) defining a symbolic rule template to represent the whole search space; (2) formulating the rule synthesis problem into symbolic constraints; and (3) solving the constraint formula and generating **all** rules that satisfy the constraints. Trace2TAP uses Z3 Theorem Prover [11] to handle the last task. Here, we focus on Step 1 and Step 2.

5.2.1 A Symbolic Rule Template. Having a symbolic rule template allows Trace2TAP to reason just about one symbolic rule, instead of many (or even infinite) concrete rules. As shown in Table 1, to form a trigger proposition using a specific variable V_{t_*}, we need to decide what constant value, represented by symbol K_{t_*}, to compare with, and what exact comparison, > or == or others, to conduct, represented by symbol $\otimes$. We then need a set of selection symbols $\boldsymbol{\lambda}_{V_{t_1}}, \boldsymbol{\lambda}_{V_{t_2}}, \boldsymbol{\lambda}_{V_{t_3}}$ to represent which one of the three candidate trigger variables and its corresponding proposition will be included in the synthesized rule.

As also shown in Table 1, we can form condition propositions in a similar way, using three sets of symbols. Note that, by definition, a TAP rule's trigger can only contain one event and hence one variable. However, a TAP rule's condition could be the conjunction of multiple propositions involving multiple variables. This difference is visualized in Table 1 and will be reflected in the symbolic constraints that we will explain later.

Once all symbols are assigned with concrete values, we get a concrete TAP rule like the example below:

Value assignment:	$\boldsymbol{\lambda}_{V_{t_1}}$:= TRUE; $\boldsymbol{\lambda}_{V_{t_2}}, \boldsymbol{\lambda}_{V_{t_3}}$:= FALSE; $\otimes_{V_{t_1}}$:= > ; $\boldsymbol{K}_{V_{t_1}}$:= 75; $\boldsymbol{\mu}_{V_{c_2}}, \boldsymbol{\mu}_{V_{c_3}}$:= TRUE; $\boldsymbol{\mu}_{V c_1}$:= FALSE; $\oplus_{V_{c_2}}$:= = ; $\boldsymbol{K}_{V_{c_2}}$:= Sunny; $\oplus_{V_{c_3}}$:= ≠ ; $\boldsymbol{K}_{V_{c_3}}$:= Off
Concrete rule:	IF V_{t_1} rises above 75 WHILE V_{c_2} is Sunny AND V_{c_3} is not Off, THEN take action $\mathbb{A}$.

5.2.2 Formulating Rule Constraints. There are two sets of constraints that we want a synthesized rule to follow: (1) the rule has to be syntax-valid and non-redundant; (2) the rule has to approximate more than a threshold portion of the instances of $\mathbb{A}$ in the trace.

Rule validity constraints. To avoid invalid rules, we require the following:

- $\forall V : R.\lambda_V \rightarrow (\neg R.\mu_V)$: Variables showing up in the trigger should not show up again in the conditions because it will contain no more information;
- $\sum_{i=1}^{3}[R.\lambda_{V_{t_i}}] == 1$: Each rule should only have one trigger;
- $\forall$ boolean $var : R.\oplus_V = \text{“ = ”} \wedge R.\otimes_V = \text{“ = ”}$: “= False” and “≠ True” mean the same thing. We force comparators for Boolean parameters to be “=”.

Rule automation constraints. Thinking about a specific action instance A that occurred at the moment t_A in the trace T, we want to formulate a constraint that reflects whether a rule R could have triggered A within the time window from $t_A - \Delta_1$ to $t_A + \Delta_2$[3] under the original context of T. We represent this constraint as the following:

$$S'[A.var] == A.val, S' = \texttt{executeTrace}\ (episode, R_1 \ldots R_m, R)$$

Here, *episode* is the sub-trace of T that spans the time window from $t - \Delta_1$ to $t + \Delta_2$, including all original events in T except for A. The function `executeTrace` outputs a formula representing the final system state after applying all the input rules to the input trace or sub-trace, which we will explain in detail in Section 5.2. The constraint above describes that at the end of the *episode*, the effect of action $\mathbb{A}$ has to be there: $S'[A.var] == A.val$.

Following this process of constraint reasoning for every single action instance A applied manually by users, we can now get the following constraint which requires the rule to approximate over a $threshold$[4] portion of all instances of action $\mathbb{A}$. Instances triggered by existing rules are not considered here because our focus in this phase is to automate manual actions. We intend the newly synthesized rule to co-exist with the existing rules. $episode_1 \ldots episode_n$ are the episodes corresponding to all of the n instances of action $\mathbb{A}$ in the original trace T:

$$\sum_{i=1}^{n}[S_i'[A.var] == A.val] \geq threshold \times n \tag{1}$$

$S_i' = \texttt{executeTrace}\ (episode_i, R_1 \ldots R_m, R)$, and “$[\cdots]$” is the indicator function of a Boolean expression.

5.2.3 `executeTrace`. As shown in Algorithm 1, `executeTrace` starts from the trace's initial state `trace.init` and keeps updating the system state by sequentially applying every external event E in the trace. For every event E, `executeTrace` first applies the direct impact of E to the system state, updating its corresponding variable $S'[E.var] := E.val$. `executeTrace` then goes through every rule R: R_1, ... R_j to see if a rule R is triggered. If so, it updates the corresponding variable: $S'[R.action.var] := R.action.val$.

The `checkRule` function listed in Algorithm 2 is used to check whether a rule R will be triggered right after an event E under a system state S. It first checks if the event E matches the trigger event of R: the variable of E needs to be selected as the trigger variable ($\exists i \in \{1, 2, 3\} : V_{t_i} == E.var$ and λ_{t_i} is true), and the post-event value setting of E needs to match that of the trigger ($E.val \otimes_{t_i} K_{t_i}$ is true). It then checks whether the state of the system S satisfies all condition propositions of R. For this checking, `checkRule` iterates through all variables that exist in the rule conditions ($V_{c_1} \cdots V_{c_3}$). For the i-th variable V_{c_i}, its condition proposition is satisfied when “$\text{cond}_i : \mu_{[V_{c_i}]} \rightarrow \left(S[V_{c_i}] \oplus_{V_{c_i}} K_{V_{c_i}}\right)$” is true. The whole rule's condition requirement is met when $\text{cond}_1 \ldots \text{cond}_3$ are all true. Finally, `checkRule` returns the conjunction of the formula `trig`, representing whether R's trigger matches E, and the formula `cond`, representing whether R's condition is satisfied by the system state S.

With all these, the `executeTrace` function will generate the symbolic formula representing the final system state S' after applying all rules R_1, ... R_j to a given trace.

[3] Δ_1 and Δ_2 are configurable. In our current prototype, they are set to 10 minutes and 5 minutes, respectively.
[4] In our current study, we set $threshold$ to 0.3.

Note that the inputs to `executeTrace` could contain not only a symbolic rule, but also concrete rules that already exist in the system. Those concrete rules can be handled as special cases of the symbolic rule with corresponding rule components containing concrete, instead of symbolic, values. Also note that it is possible for one external event to activate more than one rule. Our automation system, which we will discuss in Section 7, makes sure that those multiple activated rules will be executed one after the other with no race situation. Consequently, reasoning about multiple activated rules can be reduced to reasoning about just one rule at a time.

Input : Trace to be re-executed *trace*
Input : Rules to apply $R_1, ..., R_j$
Output: Final state S'
Function *executeTrace(trace,* $R_1 \dots R_j$*)*
  $S := trace.init$;
  for *Every E in trace.events* **do**
    if *E is external* **then**
      $S[E.var] := E.val$;
      for $r := R_1 \dots R_j$ **do**
        $triggered :=$ $checkRule(r, E, S)$;
        if *triggered* **then**
          $S[r.action.var] :=$ $r.action.val$;
  return S;

Algorithm 1: Compute the final system state after applying a set of rules to a trace

Input : A rule R
Input : An event E
Input : Current system state S
Output: A boolean *result* representing if rule is triggered
Function *checkRule(R, E, S)*
  if $\exists i : E.var = R.V_{t_i}$ **then**
    $trig := \big((E.val)R. \otimes_{t_i} (R.\boldsymbol{K}_{t_i})\big) \wedge R.\boldsymbol{\lambda}_{t_i}$;
  else
    $trig :=$ false;
  for $i := 1 \dots 3$ **do**
    $cond_i := R.\boldsymbol{\mu}_{c_i} \rightarrow \big((S[V_{c_i}])R. \oplus_{c_i} (R.\boldsymbol{K}_{c_i})\big)$;
  $cond := cond_1 \wedge \dots \wedge cond_n$;
  $result := trig \wedge cond$;
  return *result*;

Algorithm 2: Check if symbolic rule R of the format in Table 1 is triggered. We use $R.\boldsymbol{K}$, $R.\otimes$, and so on to refer to symbols in R.

5.2.4 Handling Temporal Rules. Trace2TAP also supports analyzing and generating rules with timing triggers like "exactly {*time*} ago, *var* := *value* became true and has remained so." We do not treat *time* as a symbolic variable in the symbolic rule template (Table 1) because doing so would introduce concurrency issues in symbolic re-execution and tremendously increase the complexity of our symbolic constraint solving. Instead, we pre-compute the time periods for candidate trigger variables and then adapt the symbolic rule template.

As mentioned in Section 5.1, when calculating the cross-correlation between events related to a variable V and the target action, Trace2TAP identifies the $\texttt{time}_\Delta$ used to help V achieve its highest cross-correlation. When $\texttt{time}_\Delta$ is larger than 1 second, Trace2TAP considers a timing option for any trigger proposition involving V. For example, if two out of the three trigger candidate variables have timing options 10 seconds for V_{t_1} and 60 seconds for V_{t_3}, the symbolic trigger of R would change from that in Table 1 to the following:

$$\begin{cases} \boldsymbol{\lambda}_{V_{t_1}} \Rightarrow V_{t_1} \otimes_{V_{t_1}} \boldsymbol{K}_{V_{t_1}} \\ \boldsymbol{\lambda}_{V_{t_2}} \Rightarrow V_{t_2} \otimes_{V_{t_2}} \boldsymbol{K}_{V_{t_2}} \\ \boldsymbol{\lambda}_{V_{t_3}} \Rightarrow V_{t_3} \otimes_{V_{t_3}} \boldsymbol{K}_{V_{t_3}} \\ \tilde{\boldsymbol{\lambda}}_{V_{t_1}} \Rightarrow \text{“}V_{t_1} \tilde{\otimes}_{V_{t_1}} \tilde{\boldsymbol{K}}_{V_{t_1}}\text{” has been true for exactly 10s} \\ \tilde{\boldsymbol{\lambda}}_{V_{t_3}} \Rightarrow \text{“}V_{t_3} \tilde{\otimes}_{V_{t_3}} \tilde{\boldsymbol{K}}_{V_{t_3}}\text{” has been true for exactly 60s} \end{cases}$$

$\tilde{\lambda}$, $\tilde{\otimes}$, and $\tilde{K}$ are symbols related to the timing statements. Since the time parameters (e.g., 10 seconds and 60 seconds) are concrete values, evaluating such a symbolic rule with timing triggers is very similar to that for a symbolic rule without timing triggers.

5.3 Rule Debugging and Patching

Users may not be totally satisfied with the current rules in the system and may need to manually revert some events activated by existing rules. In these cases, Trace2TAP can help generate rule patches to revert some of the automated events. These patches cover potential ways to revert some automated events by adding conditions to a rule, deleting a rule, or changing parameters in a rule. In this section, we demonstrate how Trace2TAP finds patches to add conditions to a rule.

Imagine that an existing rule R triggers an action $\mathbb{A}$ a total of m times in a trace T, where k of these m instances were reverted by the user shortly afterwards. Without loss of generality, we will denote these k instances as occurring at time t_1, t_2, ... t_k. The goal here is to add more *conditions* to the rule R, represented by the symbolic rule below, so that R would trigger its action $\mathbb{A}$ in a more selective way. We do not conduct variable selection here because the complexity is limited without modifying the rule trigger and action.

$$\text{IF } \textbf{trigger} \text{ WHILE } \textbf{conditions} \text{ AND } \begin{array}{l} \boldsymbol{\mu}_{V_{c_1}} \rightarrow V_{c_1} \oplus_{c_1} \boldsymbol{K}_{c_1} \\ \boldsymbol{\mu}_{V_{c_2}} \rightarrow V_{c_2} \oplus_{c_2} \boldsymbol{K}_{c_2} \\ \cdots \\ \boldsymbol{\mu}_{V_{c_m}} \rightarrow V_{c_m} \oplus_{c_m} \boldsymbol{K}_{c_m} \end{array} \text{ THEN take action } A.$$

Intuitively, we can evaluate whether the modified rule would still be triggered at time t_i by checking whether those extra conditions are satisfied by the system at t_i, which can be represented by the following formula:

$$T_i := \left(\boldsymbol{\mu}_{V_{c_1}} \rightarrow S_{t_i}[V_{c_1}] \oplus_{c_1} \boldsymbol{K}_{c_1}\right) \wedge \cdots \wedge \left(\boldsymbol{\mu}_{V_{c_m}} \rightarrow S_{t_i}[V_{c_m}] \oplus_{c_m} \boldsymbol{K}_{c_m}\right)$$

Then, similar to Equation 1, we can set up the constraint below. By solving it, we get all potential patches to a rule R that can keep a large proportion of R's correct behaviors ($> threshold_1$) and dismiss a certain portion of its undesirable ones ($> threshold_2$).

$$\left(\sum_{i=k+1}^{n} [T_i] \geq threshold_1 \times (n-k)\right) \wedge \left(\sum_{i=1}^{k} [\neg T_i] \geq threshold_2 \times k\right)$$

6 TRACE2TAP RULE PRESENTATION

In this section, we discuss how we designed Trace2TAP to present synthesized rule candidates intelligibly, empowering users to make intuitive and informed selections. In Section 6.1, we discuss how Trace2TAP clusters and ranks rules. In Section 6.2, we then show how Trace2TAP visualizes each rule's rationale and implications.

6.1 Clustering and Ranking

Trace2TAP organizes rule candidates in two steps. First, Trace2TAP clusters all rule candidates into groups based on the overlap in instances of manual actions they would automate, which we term a usage **context**. Trace2TAP presents one cluster at a time to the user. Second, Trace2TAP ranks the rules within each cluster using a linear scoring function that considers a combination of six features. Our goal is to aid users in identifying one or more rules within each context that match their intent. Trace2TAP's two-step approach lets users focus on one context at a time and avoid the often confounded attempt of quantitatively comparing rules across contexts.

6.1.1 Clustering. Trace2TAP clusters all rules based on which subset of user actions in the trace (context) each rule approximates. Specifically, Trace2TAP uses an n-element bit vector for each rule, with the i-th bit indicating whether the i-th instance of the target action $\mathbb{A}$ is approximated by the rule ('1') or not ('0'). For example, imagine that the user manually applied A five times. Rule 1's bit vector is 10011, while Rule 2's bit vector is 01100. This shows that Rule 1 automates different scenarios than Rule 2 and thus likely captures a different usage context.

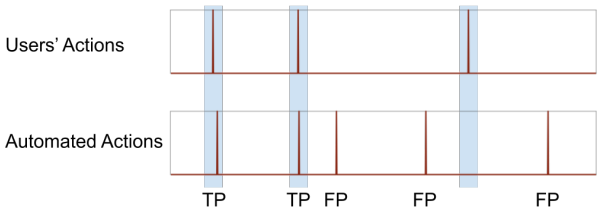


Fig. 5. Counting a rule's true positives and false positives.

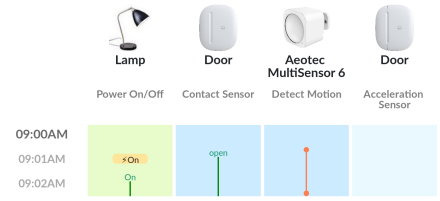


Fig. 6. A visualization of when in the trace the rule would have triggered (the "on" with the beige background).

Trace2TAP then automatically clusters all rule candidates based on the Hamming distance between their bit vectors, using the K-modes [24] unsupervised clustering algorithm. K-modes is an extension of the classic K-Means algorithm that handles categorical data spaces. Trace2TAP automatically chooses the best number of clusters based on the Silhouette score [52], limiting the maximum number of clusters to five.

6.1.2 Ranking. Since all rules within a cluster cover similar usage scenarios, users will likely want to choose at most one rule from each cluster. Trace2TAP thus ranks the rules within each cluster. Trace2TAP's ranking function considers the following six features, which we identified by manually analyzing and discussing the comprehensive list of rules synthesized during pilot deployments of Trace2TAP in the co-authors' offices:

1) *True positives (TPs):* To prioritize rules that automate many manual actions, we count the number of times the rule would have automated manual instances of the selected action in the trace (within 10 minutes before or 5 minutes after the manual action, as in Figure 5).

2) *Precision:* To prioritize rules whose automations typically correspond to manual actions, we divide the number of true positives by the total number of times the proposed rule would have been triggered in the trace.

3) *Recall:* To prioritize rules that automate many of the manual actions, we divide the number of true positives by the total number of manual action instances in the trace.

4) *Time discrepancy:* To prioritize rules that perform actions around when the user would have performed them, we average the timing difference between the rule's automated actions and the human's manual actions.

5) *Complexity:* Because shorter rules are frequently more intelligible and more generalizable, we count the number of conditions in the synthesized rule.

6) *Rule delay:* Some rules trigger only after some delay (e.g., "IF trigger has been true for 10 seconds, THEN apply action"). To minimize the time needed to take the action after an observed state change, we consider the delay in seconds, setting it to 0 for non-time-based triggers.

To weight these six factors in a principled way, we ran a formative deployment in ten offices. Note that no participants or offices in this formative deployment participated in our summative evaluation (Section 9). In this pre-study, we used Trace2TAP to synthesize rules based on a one-week trace collected from each office. We then presented 50 rules, selected using ad-hoc methods, to each participant. We asked them to label which rules they would consider selecting, and which they would not. Using this labeled data, we trained a linear scoring function consisting of the six factors above. Trace2TAP uses this linear scoring function to rank rules within each cluster. Further, Trace2TAP orders the clusters themselves based on the score of each cluster's highest-ranked rule.

6.2 Visualizing the Impact of a Prospective Rule

To help users understand the behavior of each rule candidate, Trace2TAP presents a number of metrics associated with the rule, such as the number of true positives and false positives (under different terminology). Figure 2a from Section 2 exemplifies this screen.

Trace2TAP also visualizes when the rule candidate would have triggered during the collected trace. For every moment t when the rule would have been triggered in the trace, Trace2TAP visualizes the events around t. To avoid overwhelming the user, we show only the relevant devices/sensors and only the relevant events immediately preceding and following when the rule would have automated the action. As shown in Figure 6, the column with the light green background shows the status of the target device (lamp). The two columns with darker blue backgrounds list the status of the sensors included in synthesized rules. The last column, in lighter blue, shows the status of sensors found to be related to the target action in the variable selection step, but not included in the current rule. Such sensors like "door" in Figure 6's case are often temporally or conditionally related to the target action to be automated. They help users remember the contexts at the visualized time point. In this example, the user manually turned on the lamp at 9:02am. The visualization shows that the lamp would have been turned on by the rule at 9:01am (the "on" with a beige background). The solid lines in the visualization mean that those devices/sensors were in an active state (e.g., on, open, motion detected) at the time.

7 TRACE2TAP SYSTEM IMPLEMENTATION

To let us collect rich data about Trace2TAP in ecologically valid circumstances as part of a field study in office environments, we implemented the whole Trace2TAP system. Our implementation encompasses a Samsung SmartApp and a companion web application. As mentioned in Section 1, we open-sourced our implementation.

The SmartApp serves as an intermediate layer that enables Trace2TAP to work with the devices in Samsung's vast SmartThings ecosystem. It logs all events supported by the devices, including users' interactions with smart devices and environmental information (e.g., motion, temperature, and illumination). Furthermore, the SmartApp is also responsible for interpreting every TAP rule that the user saves through the Trace2TAP web interface, executing those rules by sending commands to the relevant physical device using Samsung's SmartThings API. Trace2TAP avoids race conditions among actions triggered simultaneously by enforcing the following: (1) each rule is assigned an ID in our database; (2) only sensors (not actuations of other devices) are permitted in triggers and conditions to avoid rule chains [58]; (3) when two rules are triggered with the same event, the rule with the lower ID is run first; and (4) a newly synthesized rule always receives a higher ID than any existing rule.

The web application stores all collected traces in a database, runs the SMT-solver, and presents all user interfaces we created. In the user interface, the web application first shows users all actions that have occurred at least a threshold number of times in the trace (by default, the threshold is 4 times). As detailed in Section 2, after the user selects an action to automate, Trace2TAP synthesizes rules, clusters/ranks them, and then presents them with their accompanying visualizations. We also permit users to directly write their own TAP rules; we implemented our own interface visually approximating IFTTT [27], albeit with our expanded TAP syntax (Section 4).

8 EVALUATION METHODOLOGY

For our field study (summative evaluation), we initially recruited nine participants from the Computer Science Department of the authors' institute. Participants volunteered to have Internet-connected devices temporarily installed in their offices. We excluded from further analysis two participants who never used the installed devices during the study's time period. Among the remaining seven valid participants, two (*p3* and *p4*) are staff members without any technical expertise, while five (*p1*, *p2*, *p5*, *p6*, and *p7*) are CS faculty members. We selected this mix to gauge how both non-technical and highly technical users would interact with Trace2TAP. Non-technical staff members might not deeply understand trigger-action programming or the sensors, providing a window into Trace2TAP's ability to reach a broad audience. Technical faculty members' programming expertise would make them likely to analyze rules critically and in a manner that accounts for corner cases, providing insight into Trace2TAP's ability to synthesize appropriate rules in the nuanced circumstances of a real deployment.

Members of the research team installed Internet-connected devices (chosen by the participant) in each participant's office. These devices included various Philips Hue lighting devices, and optionally the participant's choice of an iTvanila humidifier and a Dezin electric tea kettle connected to a Samsung SmartThings smart plug. A number of sensors, including Samsung SmartThings motion sensors, Samsung SmartThings contact sensors, and Aeotec multipurpose sensors, were also installed. Depending on the size of the office, one to three Samsung SmartThings motion sensors were installed, along with one additional Aeotec multipurpose sensor, which could also be used for motion detection. These motion-related sensors were placed strategically to cover the maximum area in an office. In addition to motion detection, the Aeotec multipurpose sensor also measures illuminance, humidity, temperature, UV index, and vibration. Contact sensors were installed on all doors to detect if they were open or closed. Across the seven valid participants' offices, we installed 2 floor lamps, 10 table lamps, 3 lightstrips, 3 kettles, 1 humidifier, 8 motion sensors, 7 multi-purpose sensors, 12 buttons, and 7 door contact sensors.

The field study ran from November 2019 to March 2020. In the study, participants were asked to interact with their devices manually, which was traced by Trace2TAP. At the end of the study, we held a semi-structured interview with each participant. We introduced them to the concept of trigger-action programming and had them use Trace2TAP's interactive workflow, as described in Section 2. For each action that Trace2TAP proposed automating, we asked participants to go through the suggested rules in each cluster and select any they liked. We permitted participants to use the interface organically, without assistance. Participants were allowed to skip lower-ranked rules within a cluster, though we required they at least look at each cluster. Throughout this process, we asked participants to think aloud, as well as to explain why they decided to accept or reject particular rules.[5]

9 EVALUATION RESULTS

During the 4 months of our field study, Trace2TAP recorded 1,226,688 events in total across the installed devices, 4,405 of which were participants' manual interactions with actuators. Across the seven offices, the traces contained 18 unique actions on 9 unique devices (7 table lamps, 1 lightstrip, and a floor lamp) that Trace2TAP considered as targets for automation.[6]

For these 18 target actions, Trace2TAP synthesized 71 clusters containing 1,578 rules in total. The synthesis for most actions was completed within 30 seconds. By selectively navigating these rules using Trace2TAP's interface, participants selected 32 rules from 31 clusters to install in their offices, automating 16 out of the 18 target actions. The median rank of the 32 rules selected by participants was *second* within each cluster.

In this section, we present detailed results about the effectiveness of Trace2TAP's clustering and ranking schemes (Section 9.1 and 9.3), how important it is for rule synthesis to be *comprehensive* (Section 9.2), and what factors influenced participants' decision processes (Section 9.4).

9.1 How Effective is Clustering?

Are there different contexts under which an action occurs? Our results show that the answer is "yes." In most cases, participants need more than one rule to automate a single action to their satisfaction, with the median number of rules picked for each action being 2, and the largest number being 4. These rules automate different usage contexts of a single action.

Are we clustering the right sets of rules together? Ideally, we want each cluster to represent a unique context. Based on the results, we have achieved this goal. Among the 32 rules accepted by the participants, only two of

[5]We had planned a subsequent phase of the study to evaluate Trace2TAP's debugging features. This phase was cut short by the COVID-19 pandemic. We were only able to witness Trace2TAP's debugging feature in action anecdotally during our formative deployment (see Section 2).
[6]Occasionally, the participants used multiple devices (i.e., lights) in almost the same way. In those cases, we only covered one representative device in our interviews and skipped the others.

them came from the same cluster. Future users of Trace2TAP can simply stop checking more rules in a cluster after they accept one from that cluster.

Does clustering help users find their rules more easily? To compare the difference between using and not using clustering, we first compared the ranking of the selected rules within each cluster and globally (i.e., making a single ranked list of all rules synthesized) under the same ranking function.

As shown in Figure 7, the median rank of the selected rules within their clusters (blue bars and the blue box plot below) was 2, whereas their median rank globally was 7.5 (orange bars and the orange box plot below)—a striking difference in ranking. As visualized by the cumulative distribution function (CDF) curve in the figure, more than 80% of the selected rules are ranked in the top 5 within their clusters. However, close to half of the selected rules are not among the top 10 rules in the global ranking. Without clustering, many of these low-ranked rules may never have been viewed by participants, nor selected by them.

For further comparison, we evaluated what proportion of selected rules would have been covered after users checked the first N rules, with and without clustering. Determining the first N rules a user would check without clustering is straightforward as all rules are totally ordered without clustering. To approximately identify the first N rules a user would check with clustering, we assumed a breadth-first search style of rule navigation: first checking all rules ranked first across all clusters, then all rules ranked second across all clusters, and so on. Once the user accepted a rule in a cluster, we assumed the remainder of that cluster would be skipped. We examined the total number of rules users needed to check across all 18 actions to reach a given coverage of the 32 selected rules under the search policy, showing the result in Figure 8. Note that this would be an underestimate of the capability of clustering because Trace2TAP actually ranked clusters themselves, presenting the most promising clusters first rather than treating them equally. Even with this underestimate, Figure 8 shows that Trace2TAP's clustering can help users see more promising rules while expending the same amount of search effort. Putting all the rules synthesized for all actions together, without clustering participants would have needed to examine 386 and 501 rule candidates to find 70% and 80%, respectively, of the rules they eventually chose to install. In contrast, with clustering, they only needed to examine 203 and 274 rule candidates, respectively. Thus, clustering reduced the effort needed by half.

9.2 How Important is it for Rule Synthesis to be Comprehensive?

We examined several features of the 32 rules that were selected by participants to see how comprehensive synthesis of rule candidates needs to be. First, for every selected rule that automates action $\mathbb{A}$, we checked what proportion of $\mathbb{A}$ instances in the trace could have been approximated by this rule if it was installed at the beginning of the field study. The histogram of this automation-proportion achieved by each selected rule is shown in Figure 9. This figure highlights that 10 out of the 32 selected rules actually approximate less than 40% of their corresponding actions' instances (the threshold used in Trace2TAP is 30%), while the mean approximation proportion is only 57.2%. If we were to have raised the threshold even to 50%, almost half of the rules that participants selected would have disappeared from our rule candidates. This result supports Trace2TAP's design philosophy of being "comprehensive," one of its key advantages over prior work.

We also examined the number of conditions each of the 32 rules contains. As shown in Figure 10, rules with 1 condition are the most common (about 40%) among the 32 rules, followed by rules with no condition (close to an additional 40%). Rules with more conditions are rarer among those selected, but they do exist: 6 selected rules contain 2 conditions, while 1 selected rule contains 3 conditions. Unsurprisingly, the overall trend seems to be that participants prefer rules with fewer conditions, which are less complicated and frequently more generalizable than those with many conditions. However, the good number of zero-condition, one-condition, and two-condition rules again demonstrates that a variety of types of rules appealed to participants, so it is important for rule synthesis to be comprehensive.

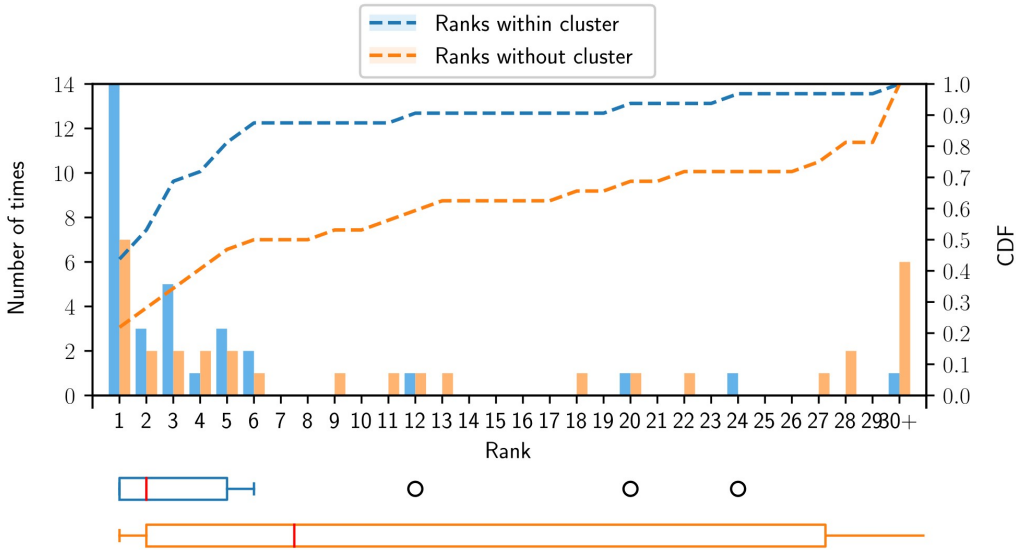


Fig. 7. Rank distribution of rules selected by participants with and without clustering. The dashed curves are CDFs.

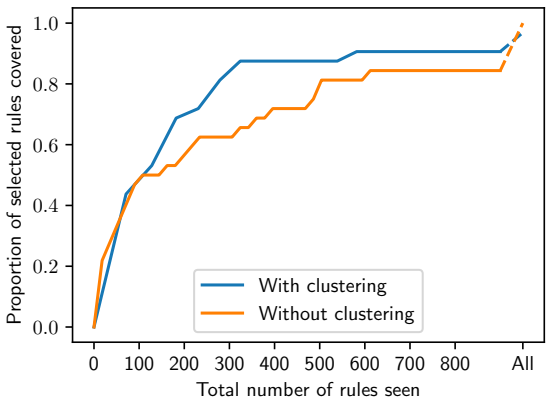


Fig. 8. Coverage of selected rules by the number seen. The x-axis is aggregated from rules for all target actions.

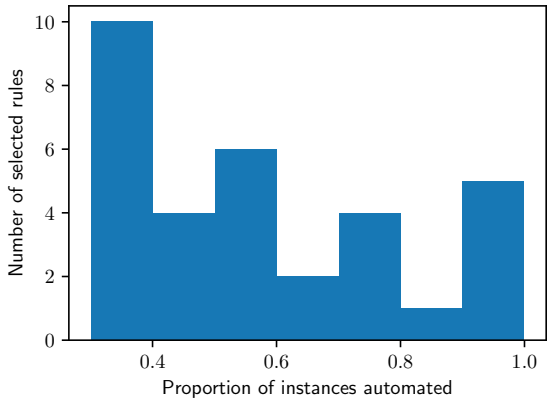


Fig. 9. The proportion of manual instances automated.

9.3 How Effective is the Ranking Function?

Trace2TAP uses its ranking function, which was discussed in Section 6.1, in two places. First, Trace2TAP uses it to rank rules within a given cluster. To see how effective the ranking function is for this purpose, we compared it with two naive ranking functions, one ranking solely based on true positives (TP) and one ranking solely based on precision (i.e., $\frac{True\ Positives}{True\ Positives+False\ Positives}$). As shown in Figure 12, Trace2TAP's current ranking function ranks the selected rules highly in each cluster (median rank 2), while the two alternative ranking functions perform much worse, offering a median rank of 6 and 4.5, respectively.

Second, Trace2TAP also uses the ranking function to decide which cluster of rules to present first. The higher a cluster's top-ranked rule scores, the earlier this cluster will be shown to the user, although we encouraged every

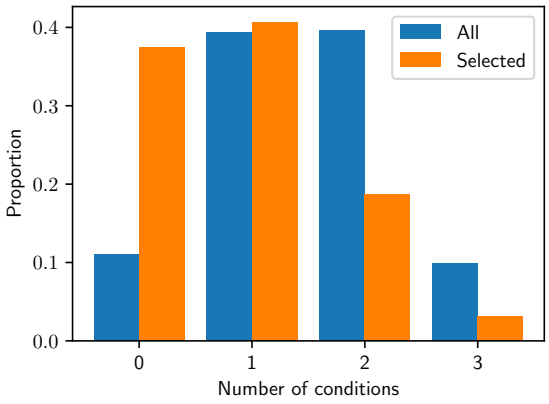


Fig. 10. The number of conditions rules selected by participants have, compared to all rules synthesized.

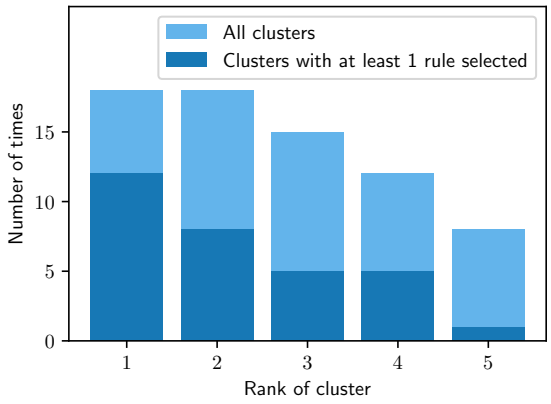


Fig. 11. The distribution of the ranks of clusters with at least one rule selected.

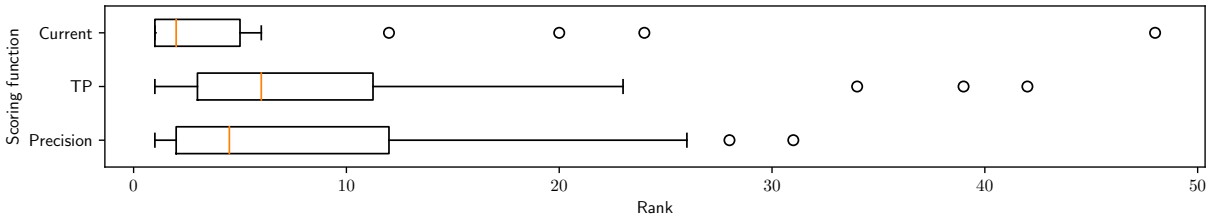


Fig. 12. The ranks within a cluster of rules selected by participants under the current scoring function, as well as under potential alternatives relying on true positives (*TP*) and precision.

participant in the field study to at least look at every cluster. Figure 11 shows the total number of clusters ranked k-th among all clusters for one action, as well as the total number of k-th ranked clusters that have at least one rule picked by the participants. As the figure highlights, participants picked rules from higher-ranked clusters more frequently than from lower-ranked clusters. Notably, participants picked rules from 12 out of 18 top-ranked clusters. At the same time, participants also selected rules regularly even from low-ranked clusters. For example, participants selected rules from 5 out of the 12 clusters ranked fourth.

9.4 Qualitative Analysis of Participants' Rule-Selection Processes

To better understand participants' approaches, we analyzed qualitative data from our semi-structured interviews.

9.4.1 General Themes behind Rule Acceptance and Rejection. To summarize common themes in the interviews regarding why the participant decided to accept or reject a rule, one researcher on the team went through the recordings and created a codebook that contains reasons why a participant accepted or rejected a rule. Specifically, reasons participants expressed for accepting a rule were: (1) *Anti-FP (false positives)*: The rule would not be triggered in scenarios where participants didn't want it to be triggered; (2) *Trace match*: The rule matched participants' past behavior; (3) *Intention match*: The rule did what participants wished to do. (4) *Short edit distance*: The rule was close to participants' intended rule. (5) *"Have a try"*: Participants would like to try the rule. (6) *Simplicity*: The rule was simple without unnecessary things involved. (7) *Good time*: The time shown in the rule was good. In contrast, reasons participants expressed for rejecting a rule were: (1) *False positives*: The rule would be triggered in undesired cases; (2) *Wrong condition*: The rule had a wrong or unnecessary condition; (3) *Missing*

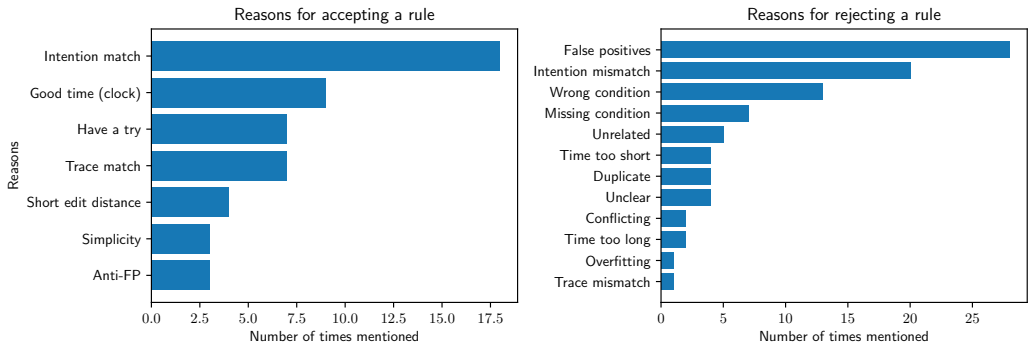


Fig. 13. The number of times participants stated the given reason for accepting/rejecting a rule in our qualitative interviews.

condition: The rule should have an additional condition; (4) *Intention mismatch*: The rule was not doing what participants wanted; (5) *Trace mismatch*: The rule did not match what participants usually did.

Using the above codebook, two researchers coded the interview data independently and then met to resolve disagreements. The result is shown in Figure 13. Several trends stand out. First, "intention match" was a dominant reason for participants to accept a rule, contributing to 18 out of 32 selected rules. "Intention mismatch" was likewise a common reason for participants to reject a rule. In comparison, more objective reasons like "trace match" and "trace mismatch" appear much less frequently. This result further demonstrates that the most important metric for a user is not how accurately a rule matches past traces, but instead how well it matches their intention. It is thus crucial to involve users in the rule-selection process. Second, participants mentioned that the rules had good timing (9 times), especially for the rules that have a time trigger (e.g. "IF it becomes 9am, THEN turn on my light"), indicating that Trace2TAP is handling temporal rules effectively. Third, "too many false positives" was a common reason for participants to reject a rule, partly because Trace2TAP rule synthesis currently does not have constraints on the number of false positives. However, as we will discuss next, naively filtering out all rules with a large number of false positives would eliminate some rules that participants selected.

9.4.2 Do Users Have Different Priorities and Concerns? We noticed from our qualitative data that participants did not reason about rules in the same way. Their reasoning for their rule selection reflects their different preferences, technical background, and even mental models for how devices should be automated. We highlight some of the salient observations below. Such results provide further evidence that rule generation should be comprehensive and should involve the human in the loop to best accommodate users' different priorities and concerns.

Technical background. With the caveat that our study was very small-scale, we noticed that participants with more background in programming tended to be more cautious, reasoning carefully about different scenarios in which a rule might be triggered. On the other hand, participants with less technical background (P3, P4) were often willing to give rules a try and generally were more open to installing new rules. While P3 and P4 selected 14 rules out of 26 clusters (53.8%), the other participants selected 18 rules out of 45 clusters (40.0%).

Personal preferences and mental models. Participants' personality sometimes influenced their rule selection. For example, P3 mentioned that she preferred to see the light being turned off in person. Otherwise she felt she needed to go back to her office to double-check. As a result, she rejected all rules that would turn off the light with a delay after she left the office (e.g., if no motion has been detected for 1 minute, turn off the light). P5 cared more about comfort than energy efficiency. She selected some rules for turning on the light, but no rules for turning off the light, saying she would be annoyed if the light turned off incorrectly.

Different participants also treated the same device differently. For example, P4 used her lamp as an "indicator," rather than a light source. She chose rules that turned on her lamp when no motion had been detected, expressing that the lamp turning on would be a good reminder of her lack of activity. In contrast, P2 and P5 used the lamp as a light source and told us that it should always be on during their work hours, regardless of whether motion was detected or not. In summary, different models and priorities for devices can lead to diversity in rule selection.

To give users the chance to find their ideal rules, we believe there are two important design considerations. First, a system should show a large variety of potential automations (i.e., TAP rules) to users. With such diversity, the system is more likely to capture different people's intent. Second, the result should be explainable—users need to understand what each program would do to make their selections. Therefore, Trace2TAP takes a comprehensive approach in synthesizing rules, also committing huge effort to summarizing and visualizing rules. In contrast, synthesizers that rely only on users' past traces or take black-box approaches are likely to miss such factors.

9.4.3 Can We Trust the Trace? We next analyzed the degree to which participants' intentions were, or were not, reflected verbatim in the trace. Our results emphasized that a home automation system should not just try to learn automations verbatim from users' past behaviors (traces). There is a bi-directional influence between users' behavior and the automation system. For example, a user can only turn off the lights before leaving the office, but an automated device can turn them off after the user leaves. It is critical for automation systems to understand that users' manual capabilities are sometimes more limited than what they truly prefer to automate.

One of the key advantages of Trace2TAP's approach over prior work is that its approach to abstraction can synthesize TAP rules that reflect orderings of events that differ from the trace. To quantify this benefit regarding timing, we examined how often the rules selected by users would cause events to occur in a different timing order from what was recorded in the trace, which we term a **timing mis-order**. Specifically, for each selected rule I, we looked back into users' traces and checked every moment when it automated a target manual action A. If this rule's trigger event occurred after A, we considered it a case of timing mis-order. For each rule, we calculated the proportion of such timing mis-orders among all instances when it automated A. For example, if the selected rule was "IF door closes THEN turn off the lamp," we checked how often the door was actually closed *after* the lamp was turned off in the trace, instead of before it. Figure 14 shows a histogram of such proportions for all 32 rules selected by participants. Over half of the selected rules have such mis-ordered cases, and 9% are almost exclusively mis-ordered. This indicates the necessity of Trace2TAP handling timing mis-orders (see Section 5).

To further understand the value of Trace2TAP synthesizing rules that effectively deviate from the trace, we measured the degree to which pattern mining approaches typical of prior work [51] could also have synthesized the TAP rules participants ultimately selected. We wrote our own implementation approximating such approaches. We also varied these approaches to roughly capture Trace2TAP's novel capabilities to handle mis-ordered events (**order-tolerant**) and triggers with a time delay, like "the door has been closed for 5 minutes" (**delay-tolerant**). Figure 15 shows that pattern mining approaches in their original form capture few of the rules participants selected. Even with these new capabilities, about one-third of rules participants selected still cannot be covered.

Causality vs. Correlation. We also noticed that when Trace2TAP synthesized TAP rules, participants were unsurprisingly looking for causality, rather than incidental correlation. Whether a correlation is incidental or reflective of some causal phenomenon is unique knowledge the human brings to the interaction, which is why Trace2TAP's approach of having the human in the loop is critical. The *trigger* should be the reason to apply the *action* in a TAP rule. However, a trace captures only correlations between events. Sometimes, the TAP rule that best fits the trace under some metrics (e.g., precision, recall) might only represent an incidental correlation rather than causality. For example, in one of our pilot studies, a temperature sensor was installed under the lamp, and Trace2TAP subsequently synthesized rules that turned on the lamp when the temperature was high. While these rules had a high precision score, they were only measuring a sensed artifact of the lamp being on, not a causal

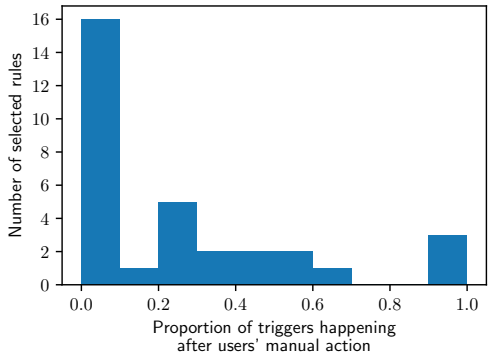


Fig. 14. Number of selected rules that do not follow event timing in the traces (timing mis-order).

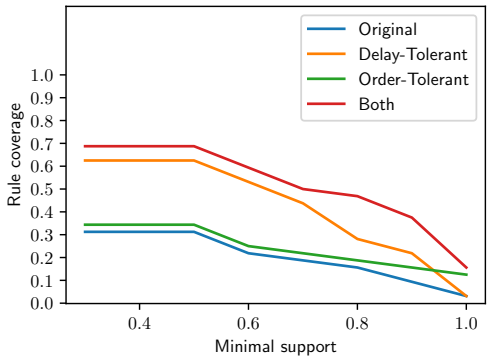


Fig. 15. The coverage of selected rules using variants of prior work's pattern mining approach, which is parameterized by the minimal support.

trigger. We believe being comprehensive in rule synthesis is important for tackling this problem because the "best" rules under a certain metric might only represent correlation, rather than causality.

False positives or not. Another reason not to rely on a rigid metric in rule synthesis is that a synthesized rule's apparent false positives relative to the trace may not actually be false positives regarding users' intent. A "false positive" only means that the rule would be triggered at a time when the user did not apply the target action in the trace. However, it does not mean the user intended not to apply the action. The user may simply have forgotten about applying the action or was unable to do so. For example, P4 mentioned that she often forgot to turn on the lamp even when she intended to do so. As a result, she selected a rule that seemed to have many false positives. Such rules would not be shown to users if we only tried to find the "best" rule under a rigid metric.

9.4.4 How to Deploy Smart Devices? It is critical to install devices, especially sensors, to maximize the amount of information collected in the trace. During our field study, P2 mentioned that it was impossible for the system to identify when he was at his conference table (a signal he hoped to use as a trigger) because no sensors were located there. P6 told us that detecting motion near the door was not as important as detecting motion at his seat. Users will need to consider their daily routines to help installers place sensors appropriately. Future work could perhaps equip the system to determine automatically whether sensors are optimally placed based on the trace.

Another issue is that it is often difficult to help users understand how sensors work. Nearly all participants raised some questions about motion sensors. Some of the motion sensors did not have a visual indicator showing whether motion had been detected. Participants thus expressed their uncertainty about the delay, sensitivity, or range of the motion sensors. The same issue applied to illuminance sensors as participants struggled to build a relationship between their perception of the brightness and the reading from the sensor. During the interview, we often needed to do some experiments with the participants to help them understand the way such sensors worked. More intuitive ways are needed to communicate to users sensors' functionalities and capabilities.

10 CONCLUSION AND FUTURE WORK

With the ubiquity of consumer smart devices, how to effectively align user intent with device automation has been a crucial open problem. In contrast to prior work that either fully relies on users to write trigger-action programs or fully relies on automated learning to infer automation models, this paper proposes a new hybrid approach that combines the respective strengths of those two methods. To accommodate for users' diverse priorities and concerns, Trace2TAP applies symbolic reasoning and SAT solving in a novel manner to search

the space of TAP rules exhaustively and synthesize a comprehensive set of program candidates automatically. It then employs a novel prioritization scheme and user interface to help users navigate and identify desired candidates. Our field study of 7 participants over 4 months of trace collection demonstrated that Trace2TAP is effective in automatically synthesizing rules that align with users' intentions, including rules that would be deemed "undesirable" by traditional metrics like precision and recall. Trace2TAP's novel prioritization scheme also helps participants navigate rules with greater efficiency than alternative schemes like global ranking. Our semi-structured interviews with the participants revealed diverse user priorities and intentions for automation. This finding, along with the principle of giving users control as rooted in ubicomp literature [4, 15, 16, 40, 60], highlights the benefits of Trace2TAP's approach in involving the user even when the system automatically synthesizes automation from observed behaviors.

We believe Trace2TAP is a starting point in combining comprehensive automation and end-user participation for smart system automation. Trace2TAP relies on a flexible framework to synthesize TAP rules and can be easily extended to meet extra constraints. This paper has shown that, by changing only the template and the constraints, one can deploy Trace2TAP as a debugger for TAP rules. Currently, Trace2TAP only considers implicit feedback from users ("the user applied some action" or "the user reverted some action"). Future work can easily extend Trace2TAP to consider users' explicit requirements. For example, the user could say "add a new condition about the brightness in the rule to make something not happen again," which can be translated to a constraint in our rule-debugging template. This functionality can increase user control and flexibility in using Trace2TAP. Moreover, due to the COVID-19 pandemic, we were unable to broadly recruit participants for our evaluation, and our results are biased toward technical users. In contrast, the intended users of Trace2TAP are non-technical. Future studies could recruit participants that better represent average users of smart home devices.

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, OAC-1835890, CNS-1764039, CNS-1563956, CNS-1514256, and IIS-1546543. We thank Bo Wang for helping to implement clock-related behaviors in our web application's backend.

REFERENCES

[1] Muhammad Raisul Alam, Mamun Bin Ibne Reaz, and Mohd Alauddin Mohd Ali. 2012. A review of smart homes - Past, present, and future. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 6 (2012), 1190–1203.

[2] Noah Apthorpe, Yan Shvartzshnaider, Arunesh Mathur, Dillon Reisman, and Nick Feamster. 2018. Discovering smart home Internet of Things privacy norms using contextual integrity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 2 (2018), 59:1–59:23.

[3] Asier Aztiria, Juan Carlos Augusto, Rosa Basagoiti, Alberto Izaguirre, and Diane J. Cook. 2012. Discovering frequent user–environment interactions in intelligent environments. *Personal and Ubiquitous Computing* 16, 1 (2012), 91–103.

[4] Louise Barkhuus and Anind K. Dey. 2003. Is context-aware computing taking control away from the user? Three levels of interactivity examined. In *Proceedings of Ubicomp*.

[5] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. 2019. How users interpret bugs in trigger-action programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

[6] Julia Brich, Marcel Walch, Michael Rietzler, Michael Weber, and Florian Schaub. 2017. Exploring end user programming needs in home automation. *ACM Transactions on Computer-Human Interaction* 24, 2 (2017), 11.

[7] Lei Bu, Wen Xiong, Chieh-Jan Mike Liang, Shi Han, Dongmei Zhang, Shan Lin, and Xuandong Li. 2018. Systematically ensuring the confidence of real-time home automation IoT systems. *ACM Transactions on Cyber-Physical Systems* 2, 3 (2018), 22.

[8] Yi-Shyuan Chiang, Ruei-Che Chang, Yi-Lin Chuang, Shih-Ya Chou, Hao-Ping Lee, I-Ju Lin, Jian-Hua Jiang Chen, and Yung-Ju Chang. 2020. Exploring the design space of user-system communication for smart-home routine assistants. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

[9] Meghan Clark, Mark W. Newman, and Prabal Dutta. 2017. Devices and data and agents, oh my: How smart home abstractions prime end-user mental models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 3 (2017), 44.

[10] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. Empowering end users in debugging trigger-action rules. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.
[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems*.
[12] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.
[13] Luigi De Russis and Alberto Monge Roffarello. 2018. A debugging approach for trigger-action programming. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*.
[14] Anind K. Dey. 2001. Understanding and using context. *Personal and Ubiquitous Computing* 5, 1 (2001), 4–7.
[15] Anind K. Dey and Alan Newberger. 2009. Support for context-aware intelligibility and control. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.
[16] W. Keith Edwards and Rebecca E. Grinter. 2001. At home with ubiquitous computing: Seven challenges. In *Proceedings of Ubicomp*.
[17] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. 2017. Personalization of context-dependent applications through trigger-action rules. *ACM Transactions on Computer-Human Interaction* 24, 2 (2017), 14.
[18] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. 1996. *Algorithms for the satisfiability (SAT) problem: A survey*. Technical Report. Cincinnati University Department of Electrical and Computer Engineering.
[19] Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*.
[20] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking access control and authentication for the home internet of things (IoT). In *Proceedings of the 27th USENIX Security Symposium*.
[21] Weijia He, Jesse Martinez, Roshni Padhi, Lefan Zhang, and Blase Ur. 2019. When smart devices are stupid: Negative experiences using home smart devices. In *Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW)*.
[22] Home Assistant. 2020. https://www.home-assistant.io/docs/automation/.
[23] Justin Huang and Maya Cakmak. 2015. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.
[24] Zhexue Huang. 1998. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data Mining and Knowledge Discovery* 2, 3 (1998), 283–304.
[25] Matthew Hughes. 2018. Mozilla's new Things Gateway is an open home for your smart devices. TheNextWeb. https://thenextweb.com/dd/2018/02/07/mozillas-things-gateway-wants-your-smart-home-devices-to-play-nice/
[26] Jan Humble, Andy Crabtree, Terry Hemmings, Karl-Petter Åkesson, Boriana Koleva, Tom Rodden, and Pär Hansson. 2003. "Playing with the bits" User-configuration of ubiquitous domestic environments. In *Proceedings of Ubicomp*.
[27] IFTTT. 2020. https://ifttt.com.
[28] Insider Intelligence. 2020. How IoT devices & smart home automation is entering our homes in 2020. Business Insider. https://www.businessinsider.com/iot-smart-home-automation.
[29] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. 2010. A simple inductive synthesis methodology and its applications. *ACM Sigplan Notices* 45, 10 (2010), 36–46.
[30] Timo Jakobi, Gunnar Stevens, Nico Castelli, Corinna Ogonowski, Florian Schaub, Nils Vindice, Dave W. Randall, Peter Tolmie, and Volker Wulf. 2018. Evolving needs in IoT control and accountability: A longitudinal study on smart home intelligibility. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 4 (2018), 171:1–171:28.
[31] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd International Conference on Software Engineering*.
[32] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*.
[33] Thorin Klosowski. 2016. Automation showdown: IFTTT vs Zapier vs Microsoft Flow. LifeHacker. http://lifehacker.com/automation-showdown-ifttt-vs-zapier-vs-microsoft-flow-1782584748
[34] Joohyun Lee, Kyunghan Lee, Euijin Jeong, Jaemin Jo, and Ness B. Shroff. 2016. Context-aware application scheduling in mobile systems: What will users do and not do next?. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.
[35] Nat Levy. 2017. Microsoft updates IFTTT competitor Flow and custom app building tool PowerApps. GeekWire. https://www.geekwire.com/2017/microsoft-updates-ifttt-competitor-flow-custom-app-building-tool-powerapps/
[36] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically debugging IoT control system correctness for building automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*.
[37] Brian Y. Lim and Anind K. Dey. 2009. Assessing demand for intelligibility in context-aware applications. In *Proceedings of the 11th International Conference on Ubiquitous Computing*.

[38] Marco Manca, Fabio Paternò, Carmen Santoro, and Luca Corcella. 2019. Supporting end-user debugging of trigger-action rules for IoT applications. *International Journal of Human-Computer Studies* 123 (2019), 56–69.
[39] Sarah Mennicken, David Kim, and Elaine May Huang. 2016. Integrating the smart home into the digital calendar. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.
[40] Sarah Mennicken, Jo Vermeulen, and Elaine M. Huang. 2014. From today's augmented houses to tomorrow's smart homes: New directions for home automation research. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.
[41] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An empirical characterization of IFTTT: Ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference*.
[42] Bryan Minor, Janardhan Rao Doppa, and Diane J. Cook. 2015. Data-driven activity prediction: Algorithms, evaluation methodology, and applications. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 805–814.
[43] Bryan David Minor, Janardhan Rao Doppa, and Diane J. Cook. 2017. Learning activity predictors from sensor data: Algorithms, evaluation, and applications. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (2017), 2744–2757.
[44] Walt Mossberg. 2014. SmartThings automates your house via sensors, app. Recode. https://www.vox.com/2014/1/28/11622774/smartthings-automates-your-house-via-sensors-app
[45] Alessandro A. Nacci, Bharathan Balaji, Paola Spoletini, Rajesh Gupta, Donatella Sciuto, and Yuvraj Agarwal. 2015. Buildingrules: A trigger-action based system to manage complex commercial buildings. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.
[46] Chandrakana Nandi and Michael D. Ernst. 2016. Automatic trigger generation for rule-based smart homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*.
[47] Robert P. Nix. 1985. Editing by example. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 4 (1985), 600–621.
[48] openHAB. 2020. https://www.openhab.org/.
[49] Mitali Palekar, Earlence Fernandez, and Franziska Roesner. 2019. Analysis of the susceptibility of smart home programming interfaces to end user error. In *Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW)*.
[50] Xin Qi, Qing Yang, David T. Nguyen, and Gang Zhou. 2013. Context-aware frame rate adaption for video chat on smartphones. In *Adjunct Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.
[51] Parisa Rashidi and Diane J. Cook. 2009. Keeping the resident in the loop: Adapting the smart home to the user. *IEEE Trans. Systems, Man, and Cybernetics, Part A* 39, 5 (2009), 949–959.
[52] Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65.
[53] Maureen Schmitter-Edgecombe. 2015. Automated clinical assessment from Smart home-based behavior data. *IEEE Journal of Biomedical and Health Informatics* (2015).
[54] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[55] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *Proceedings of the 26th International Conference on World Wide Web*.
[56] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.
[57] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.
[58] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.
[59] Jong-bum Woo and Youn-kyung Lim. 2015. User experience in do-it-yourself-style smart homes. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.
[60] Rayoung Yang and Mark W. Newman. 2013. Learning from a learning thermostat: Lessons for intelligent systems for the home. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.
[61] Lana Yarosh and Pamela Zave. 2017. Locked or not?: Mental models of IoT feature interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.
[62] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: Synthesizing and repairing trigger-action programs using LTL properties. In *Proceedings of the 41st International Conference on Software Engineering*.
[63] Sha Zhao, Julian Ramos, Jianrong Tao, Ziwen Jiang, Shijian Li, Zhaohui Wu, Gang Pan, and Anind K. Dey. 2016. Discovering different kinds of smartphone users through their application usage behaviors. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.