THE UNIVERSITY OF CHICAGO

END-USER PROGRAMMING IN SMART HOMES WITH TRIGGER-ACTION PROGRAMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
LEFAN ZHANG

CHICAGO, ILLINOIS
AUGUST 11, 2022

Copyright © 2022 by Lefan Zhang

All Rights Reserved

TABLE OF CONTENTS

LIST OF FIGURES v

LIST OF TABLES vii

ACKNOWLEDGMENTS viii

ABSTRACT ix

1 INTRODUCTION 1

2 AUTOTAP: SYNTHESIZING AND REPAIRING TRIGGER-ACTION PROGRAMS USING LTL PROPERTIES 5
- 2.1 Introduction 5
- 2.2 User Study 1: Mapping Desired Properties 10
- 2.3 AutoTap property-specification interface 12
- 2.4 AutoTap TAP synthesis 14
  - 2.4.1 Step 1: Model Construction 16
  - 2.4.2 Step 2: Patching the Automaton 19
  - 2.4.3 Step 3: TAP Synthesis 21
- 2.5 Evaluation 25
  - 2.5.1 User Study 2: Specifying Rules vs. Specifying Properties 25
  - 2.5.2 TAP Program Synthesis 29
  - 2.5.3 TAP Program Fixing 30
  - 2.5.4 Handling Multiple Properties 30
- 2.6 Conclusions 31

3 TRACE2TAP: SYNTHESIZING TRIGGER-ACTION PROGRAMS FROM TRACES OF BEHAVIOR 32
- 3.1 Introduction 32
- 3.2 End-to-End Example of Trace2TAP and its User Experience 36
- 3.3 Definitions and Terminology 40
- 3.4 Trace2TAP Rule Synthesis Algorithms and Procedure 42
  - 3.4.1 Rule Synthesis: Variable Selection 43
  - 3.4.2 Rule Synthesis: Symbolic Constraint Solving 46
  - 3.4.3 Rule Debugging and Patching 52
- 3.5 Trace2TAP rule presentation 54
  - 3.5.1 Clustering and Ranking 54
  - 3.5.2 Visualizing the Impact of a Prospective Rule 56
- 3.6 Trace2TAP System Implementation 57
- 3.7 Evaluation Methodology 58
- 3.8 Evaluation Results 60
  - 3.8.1 How Effective is Clustering? 60

3.8.2 How Important is it for Rule Synthesis to be Comprehensive? 62
3.8.3 How Effective is the Ranking Function? 64
3.8.4 Qualitative Analysis of Participants' Rule-Selection Processes 65
3.9 Conclusion 71

4 RELATED WORK 73
4.1 Trigger Action Programming (TAP) in Smart Spaces 73
4.2 TAP program bug-detection and fixing 74
4.3 Program synthesis using formal methods 74
4.4 Property-specification interfaces 75
4.5 Automating Smart Spaces From Traces 75
4.6 Program Synthesis with Constraint Solving 76
4.7 Context-Aware Computing 77

5 SUMMARY 79

LIST OF FIGURES

2.1 A (buggy) TAP *rule.* . . . 6
2.2 A proposed TAP *property.* . . . 6
2.3 The TAP rule (a) cannot guarantee the property (b). . . . 6
2.4 An overview of AutoTap, which takes user-specified properties and (optionally) user-specified TAP rules to automatically generate a set of TAP rules that satisfy the properties. . . . 8
2.5 Templates in AutoTap's property-specification UI. . . . 14
2.6 AutoTap approach vs. straw-man approach . . . 15
2.7 Transition system for **RAIN** and a **Win**dow. Statements in parentheses are Atomic Propositions held in each state. . . . 17
2.8 LTL property: $\neg\mathbf{G}(RAIN.on \rightarrow Win.closed)$ . . . 18
2.9 Device system: Window + RAIN . . . 18
2.10 Büchi Automata of our running example. . . . 18
2.11 Combined Büchi Automaton of the running example. (The top is the original. The bottom is after adding a rule.) . . . 20
2.12 Device automaton (a) changed to (b) by adding a rule. . . . 21
2.13 Generalization of adding TAP rules. . . . 23
2.14 Correctness of properties and rules by task. P-values are from Holm-corrected χ^2 tests comparing the proportion of statements correct when written using rules versus properties. . . . 27
3.1 User comes to work. . . . 37
3.2 User temporarily leaves the office. . . . 37
3.3 User closes the door for work. . . . 37
3.4 User goes home. . . . 37
3.5 Visualizations of different use cases in the trace collected from an example office occupant. . . . 37
3.6 Rule 1 in Cluster 1 for turning off the lamp. . . . 38
3.7 All rules in Cluster 3, shown on demand. . . . 38
3.8 An example of Trace2TAP's UI for showing proposed rules to the occupant. . . . 38
3.9 The patch Trace2TAP suggested to modify Rule 1 from Cluster 1 based on the user reverting automations. . . . 40
3.10 Calculating how a variable *var* is related to *target_action*. . . . 44
3.11 Counting a rule's true positives and false positives. . . . 56
3.12 A visualization of when in the trace the rule would have triggered (the "on" with the beige background). . . . 56
3.13 Rank distribution of rules selected by participants with and without clustering. The dashed curves are CDFs. . . . 62
3.14 Coverage of selected rules by the number seen. The x-axis is aggregated from rules for all target actions. . . . 63
3.15 The proportion of manual instances automated. . . . 63
3.16 The number of conditions rules selected by participants have, compared to all rules synthesized. . . . 63

3.17 The distribution of the ranks of clusters with at least one rule selected. 63
3.18 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. 64
3.19 The number of times participants stated the given reason for accepting/rejecting a rule in our qualitative interviews. 66
3.20 Number of selected rules that do not follow event timing in the traces (timing mis-order). 70
3.21 The coverage of selected rules using variants of prior work's pattern mining approach, which is parameterized by the minimal support. 70

LIST OF TABLES

2.1 AutoTap's property templates. **G**, **F**, **X**, and **W** are "always Globally", "eventually in the Future", "neXt", and "Weakly until" LTL operators. *state* is a user-specified atomic proposition or its negation. # and $*$ relate to timing (Sec. 2.4.1). 11
2.2 How AutoTap fixes buggy TAP programs. Subscripts are the # of cases AutoTap patches revert the mutation. 30

3.1 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. 47

ACKNOWLEDGMENTS

ABSTRACT

End-user programming on Internet of Things (IoT) smart devices enables end-users without programming experience to automate their homes. Trigger-action programming (TAP), supported by several smart home systems [12, 32, 36, 44, 58, 62], is a common approach for such end-user programming. However, it can be hard for end-users to correctly express their intention in TAP [7, 80] even under some daily automation scenarios.

This thesis introduces our efforts to enhance end-users' trigger-action programming experience. We believe that help from automated tools can be provided to users. Across several projects, we helped end-users in all stages of TAP's life cycle including TAP creation, testing and refinement. Throughout these projects, automated tools communicate with end-users with different inputs, from their manual behaviors in their daily lives to high-level safety properties that they think should hold.

We developed AutoTap, a system that lets novice users easily specify desired properties for devices and services. AutoTap translates these properties to linear temporal logic (LTL). Then it both automatically synthesizes property-satisfying TAP rules from scratch and repairs existing TAP rules [80]. We also created Trace2TAP, a novel method for automatically synthesizing TAP rules from users' past behaviors. Given that end-users vary in their automation priorities, and sometimes choose rules that seem less desirable by traditional metrics like precision and recall, Trace2TAP comprehensively synthesizes TAP rules and brings humans into the loop during automation [81]. Lastly, we designed TapDebug, a system that automatically fix TAP rules with user-specified behavioral feedback either identified from their device usage history or explicitly specified by themselves through our novel interface. In the TapDebug study, we conducted an empirical user study to discover obstacles along the TAP debugging process and evaluated how well TapDebug's automated tool helped users overcome them.

CHAPTER 1

INTRODUCTION

Internet of Things (IoT) smart devices have become common in users' homes [39]. Such devices can communicate with each other, sense environment contexts/user behaviors, and react accordingly. This enables opportunities for automating end-users' homes and improving their daily life. End-user programming enables end-users without programming experience to configure such automation systems.

A popular approach of such end-user programming is Trigger-action programming (TAP), which is supported by IFTTT [44], Mozilla's Things Gateway [36], Samsung SmartThings [58], Microsoft Flow [44], OpenHab [62], Home Assistant [32], Ripple [12], Zapier [44], and others. In TAP, people create event-driven rules in the form "IF a **trigger** occurs, THEN perform an **action**". For example, "IF it starts raining, THEN turn the lights blue". We have two observations for trigger-action programming:

1) End-users need to be involved in creating home automation. Some may argue that end-user programming can be skipped for home automation, as we can use statistical or machine learning (ML) techniques to build automated predictors of end-users behaviors. Such predictors train models that best fit users' past behaviors based on metrics like precision and recall. Unfortunately, such approaches does not match each user's intentions, priorities and concerns. On the one hand, there is not single best metric that fits every user. Some may care more about one context than another, and others may focus on automation precision or generalization. Users need to be involved in generation of automation. On the other hand, the intended automation may not even exist in users' past behavior. For example, a user may want the light to turn off after they leave the kitchen, yet the location of the light switch means that the system will always observe that the turn-off action happens before leaving the kitchen. We cannot guarantee models learnt from users' past behavior always accurate. To conclude, end-users still need to be involved to develop home automation through end-user

programming (like TAP).

2) Writing TAP can become hard in complex device-automation scenarios. While having end-users write TAP excels at simple scenarios, it often gets hard in complex and nuanced device-automation scenarios. Brankenbury et al. [7] discovered that end-users suffer from bugs in TAP related to control-flow confusion, wrong timing, inaccurate user expectation, and so on. Our study [80] also showed that end-users did not perform well in writing correct TAPs even for tasks appearing in daily life. Only giving end-users the TAP interface is far from a perfect end-user programming experience for them.

In this thesis, we present ways to tackle the above two issues by automating end-users experience of creating , testing, and revising trigger-action programs. While traditional trigger-action programming only shows users a plain rule-creating interface, we attempt to communicate with them in more comprehensive ways. In terms of inputs from users, we can use high-level system properties, users' history in their home and so on. In terms of outputs to users, potential system's behavioral changes of TAP patches and visualization of users' past behaviors can be shown. Here, we list our efforts exploring ways to automate trigger-action programming.

AutoTap We developed AutoTap, a system that lets novice users easily specify desired properties for their devices, and creates new TAP rules/fixes existing TAP rules accordingly. AutoTap translates these properties to linear temporal logic (LTL) and both automatically synthesizes property-satisfying TAP rules from scratch and repairs existing TAP rules. AutoTap was designed based on a user study about properties users wish to express. Through a second user study, it was shown novice users made significantly fewer mistakes when expressing desired behaviors using AutoTap than using TAP rules. Our experiments showed that AutoTap is a simple and effective option for expressive end-user programming [80].

Trace2TAP Another approach to automate end-users' TAP process is Trace2TAP, a novel method for automatically synthesizing TAP rules from traces (time-stamped logs of sensor readings and manual actuations of devices). Trace2TAP uses a novel algorithm that uses symbolic reasoning and SAT-solving to synthesize TAP rules from traces. It deploys a clustering/ranking system and visualization interface to intelligibly present the synthesized rules to users. Trace2TAP was evaluated through a field study in seven 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 [81].

TapDiff (led by another PhD student) While AutoTap and Trace2TAP focus mostly on getting information from users and transform it to TAP rules (input from users), we also designed TapDiff, a set of interfaces to show differences of TAPs to end-users (output to users). In the TapDiff project, several user interfaces and underlying algorithms were designed to highlight differences between trigger-action programs. The novel interfaces better enabled participants to select trigger-action programs matching intended goals in complex, yet realistic, situations that proved very difficult when using traditional interfaces showing syntax differences [84, 83]. Since this project was led by another PhD student, we do not go through details of this work in the proposal.

TapDebug TapDebug is a system that focuses on helping end-users debug their trigger-action programs (revising phase). TapDebug consists of multiple automated tools helping end-users in all stages of TAP debugging. It identifies automated behaviors that were problematic either by explicitly asking end-users or by analyzing the history of the smart devices. Based on such feedback, it automatically suggests ways to modify current TAP rules that

solve the issue. During evaluation of TapDebug, we conducted an empirical study where participants went through the whole TAP debugging process either with or without our automated help. From the study, we discovered obstacles end-users might face while debugging TAPs, and showed our tools' performance in helping end-users overcome the obstacles.

CHAPTER 2

AUTOTAP: SYNTHESIZING AND REPAIRING TRIGGER-ACTION PROGRAMS USING LTL PROPERTIES

2.1 Introduction

End-user programming enables users without programming experience to customize and automate systems. An approach that is particularly popular for automating IoT smart devices and online services is trigger-action programming (TAP), which is supported by IFTTT [44], Mozilla's Things Gateway [36], Samsung SmartThings [58], Microsoft Flow [44], OpenHab [62], Home Assistant [32], Ripple [12], Zapier [44], and others. Some of these TAP services are widely used [74, 55].

In TAP, users create event-driven **rules** of the form "IF a **trigger** occurs, THEN perform an **action**." For example, "IF a sad song comes on THEN turn the lights blue." Unfortunately, while novice users are able to successfully express many automation behaviors using TAP interfaces [73], attempts to express more complex, yet commonly desired, behaviors often contain bugs [33, 79, 59, 8, 7]. These bugs encompass timing errors [33], issues with control flow [76], conflicting behaviors [59], and incorrect user expectations [7]. As a result, an important open question is how to help users with no programming experience, and therefore no debugging experience, *correctly* express their wide variety of desired behaviors in TAP. Otherwise, users will encounter frustration and experience safety threats [72] from buggy TAP rules.

For example, imagine the simple and sensible desire to keep the window closed when it is raining. With current interfaces, a user might create the straightforward TAP rule "IF it begins to rain THEN close the window" (Figure 2.1). Unfortunately, this rule is insufficient. For example, while it is raining, a different rule might be triggered and open the window, or an oblivious person might open the window manually. To fully express this desire therefore

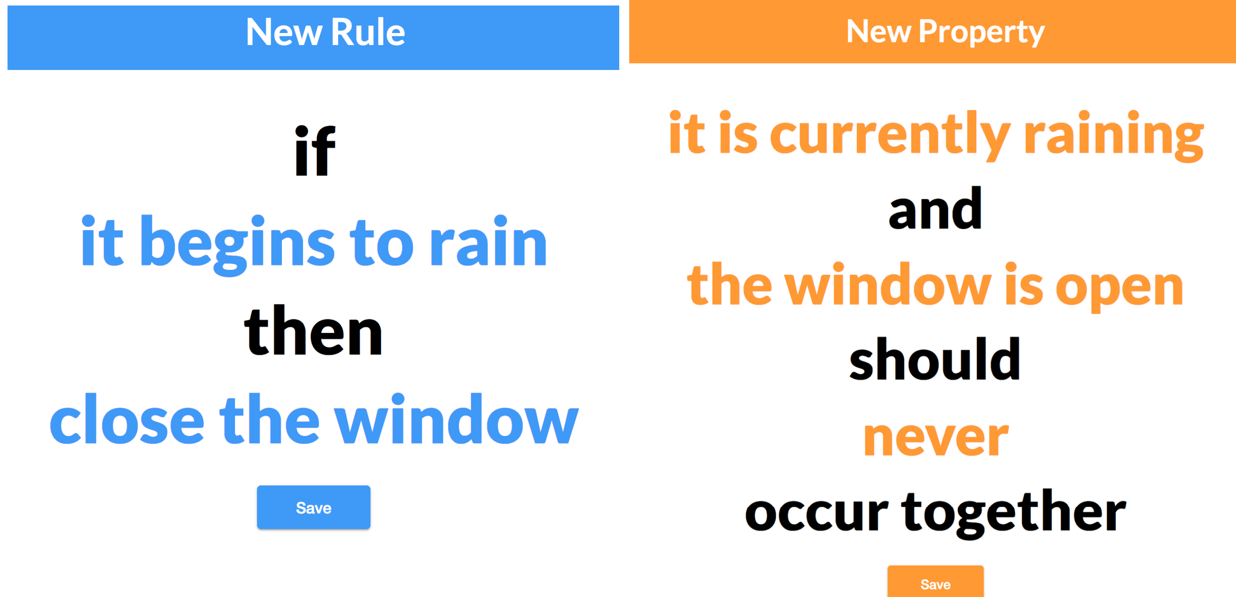

Figure 2.1: A (buggy) TAP *rule*.

Figure 2.2: A proposed TAP *property*.

Figure 2.3: The TAP rule (a) cannot guarantee the property (b).

requires a complex set of rules.

To address this open question, we present AutoTap, a system that provides easy end-user programming for smart devices and online services with fewer chances for human mistakes. AutoTap expands TAP to allow users to specify through graphical interfaces not only rules, but also **properties** about the system that should always be satisfied. For example, from the running example, one could express the desired property that "it is currently raining" and "the window is open" should never occur together (Figure 2.2). In other words, instead of requiring users to explicitly write event-driven rules defining how devices should behave, we let them simply specify what properties the system must satisfy.

If no relevant rules are provided, AutoTap automatically synthesizes property-satisfying TAP rules from scratch. For example, given the property in Figure 2.2, AutoTap will automatically synthesize two TAP rules to satisfy this property:

- IF *it begins to rain* WHILE *the window is open* THEN *close the window*
- IF *the window opens* WHILE *it is raining* THEN *close the window*

If initial rules are provided alongside the desired property, AutoTap will automatically check these rules and, if necessary, repair them to prevent the system from violating the property. AutoTap thus minimizes the opportunity for TAP mistakes. The following two key components of AutoTap work together to achieve the above functionality:

A novel property-specification interface

The key goal of TAP is to empower novice users without programming knowledge to automate and customize their devices and services. AutoTap therefore needs an interface that is both (a) **expressive**, allowing users to specify most of their desired properties for smart-device systems, and (b) **easy-to-use**, requiring minimal training for non-technical home users to use correctly.

To this end, we first conducted an online user study in which 71 current users of smart devices each provided (in free text) ten properties they would want their devices to satisfy (Section 2.2). We qualitatively coded their responses, finding that nearly all the desired properties followed one of seven templates. Subsequently, we implemented a graphical, click-only interface that mirrors the design of popular TAP rule-specification interfaces [44]. This interface enables users to specify properties following these seven templates without requiring any text input. AutoTap then directly translates properties specified in this interface to formulas in linear temporal logic (*LTL*) that can be used by AutoTap's other components (Section 2.3). While prior work has proposed interfaces for property specification [51], no prior efforts fully satisfy our requirements in the unique context of smart-device systems (Section 4.2).

Novel synthesis techniques for TAP rules

We want all programs synthesized by AutoTap to be (a) **property-compliant**, guaranteeing the programmed devices satisfy the specified properties; (b) **accommodating**, not

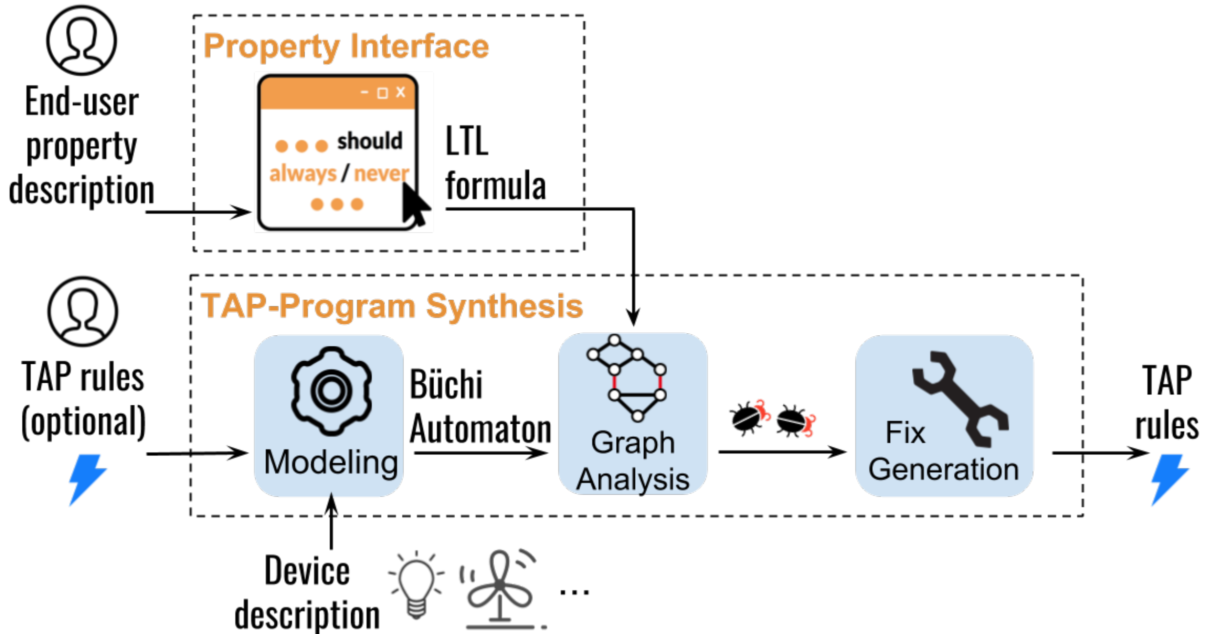

Figure 2.4: An overview of AutoTap, which takes user-specified properties and (optionally) user-specified TAP rules to automatically generate a set of TAP rules that satisfy the properties.

disabling any device behaviors that originally satisfy the properties — crucial for human-centric systems; and (c) **valid**, following the syntax of TAP rules and physical constraints of smart devices. For example, given the property in Figure 2.2, generating only one of the two TAP rules presented earlier is accommodating, yet non-compliant. Generating TAP rules that prevent the window from ever opening even in sunny weather is compliant, yet not accommodating. Generating TAP rules that prevent rain is impossible, and therefore not valid.

To achieve these goals, AutoTap takes three steps, as shown in Figure 2.4. First, it automatically builds a Büchi Automaton to formally model desired properties and the smart-device system itself, including any existing TAP rules. At this step, the novel techniques we introduce simplify models and properly represent time-related properties (Section 2.4.1).

Second, AutoTap leverages a unique feature shared by all LTL safety properties to design a simple algorithm that identifies Büchi Automaton edges whose removal guarantees the

compliant and *accommodating* goals of synthesis (Sec. 2.4.2).

Third, AutoTap designs an algorithm to systematically synthesize *valid* new TAP rules or rule changes to remove Automaton edges identified above, while making a best effort to keep rules simple and thus intelligible for users (Section 2.4.3).

These techniques are general. They are not limited to any specific patch template. They apply to any LTL safety property, not just those that can be expressed using AutoTap's current property-creation interface. Furthermore, while our interface design focuses on smart devices, the same techniques apply to online services, such as the hundreds IFTTT supports [55].

These techniques are also novel. We cannot use previously proposed synthesizers [10, 64, 47], which do not satisfy the requirements discussed above in the unique context of smart-device systems (Section 4.2). A small but quickly growing literature has begun to apply formal methods to TAP [49, 9, 60, 11]. Our techniques move beyond this work in both the target and the solution. Some of this work only aims to detect property violations [11], while others only repair *existing* rules by editing or adding conditions [49, 9] or triggers [60]. Our techniques are the first to also synthesize *new* rules from scratch and to provide the accommodating guarantees, not disabling any device behaviors that originally satisfy the desired properties — a crucial feature for human-centric systems that fundamentally cannot be provided using the fixing-by-counterexample approach of previous work [21, 49].

Our evaluation of AutoTap includes several parts (Section 2.5). We conducted a second user study in which 78 participants were randomly assigned to use either a traditional TAP rule interface or our AutoTap property interface. They used their assigned interface to express 7 behaviors randomly assigned from a larger set of 14. For *all* 14 behaviors, a larger fraction of participants using the AutoTap property interface correctly expressed the behavior than those using the traditional TAP rule interface. We also benchmarked AutoTap's performance, synthesizing TAP rules from scratch using the sets of correct properties

collected in our study. AutoTap successfully generated patches for 157 of these 158 sets.

To encourage replication and adoption, we are open-sourcing the code for both AutoTap and our rule- and property-specification interfaces. We are also releasing the anonymized data from our two user studies (with the permission of both our IRB and participants) and our full survey instruments. All of these are available at https://www.github.com/zlfben/autotap.

2.2 User Study 1: Mapping Desired Properties

To understand what types of properties users commonly desire for smart devices, we conducted an online user study.

Methodology: We designed a survey asking people who had experience with IoT smart devices in their own homes to write free-text properties they would want their devices and home to satisfy. Specifically, we asked them to write "statements about internet-connected household devices that you believe should be effective at all times, with only occasional exceptions, if any." To encourage diversity, we asked participants to imagine their house was filled with 27 smart devices we listed. We asked for ten statements, preferably five that should always be true and five that should never be true in their smart home.

We recruited participants on Amazon's Mechanical Turk who reported having an internet-connected household IoT device and living in the USA. We compensated $5 for the study, which also included a section on experiences with buggy behaviors in smart homes that is outside this paper's scope.

Through qualitative coding, we analyzed and grouped these free-text desired properties into templates. Members of the research team read through responses and iteratively proposed templates. Two coders then independently categorized each response ($\kappa = 0.62$) and met to resolve discrepancies.

To encourage complex and diverse properties, we randomly assigned half of participants

Table 2.1: AutoTap's property templates. **G**, **F**, **X**, and **W** are "always Globally", "eventually in the Future", "neXt", and "Weakly until" LTL operators. *state* is a user-specified atomic proposition or its negation. # and $*$ relate to timing (Sec. 2.4.1).

Property Type	Input Template	LTL Formula
One-State Unconditional	[*state*] should [*always*] be active [*state*] should [*never*] be active	$\mathbf{G}(state)$ $\neg\mathbf{F}(state)$
One-Event Unconditional	[event] should [*never*] happen	$\neg\mathbf{F}(@event)$
One-State Duration	[*state*] should [*always*] be active for more than [*time*] [*state*] should [*never*] be active for more than [*time*]	$\mathbf{G}(state \rightarrow (state\mathbf{W}time * state))$ $\neg\mathbf{F}(time * state)$
Multi-State Unconditional	[$state_1, ..., state_n$] should [*always*] occur together [$state_1, ..., state_n$] should [*never*] occur together	$\neg\mathbf{F}(!(state_1 \leftrightarrow ... \leftrightarrow state_n))$ $\neg\mathbf{F}(state_1 \wedge ... \wedge state_n)$
State-State Conditional	[*state*] should [*always*] be active while [$state_1, ..., state_n$] [*state*] should [*never*] be active while [$state_1, ..., state_n$]	$\mathbf{G}((state_1 \wedge ... \wedge state_n) \rightarrow state)$ $\neg\mathbf{F}(state_1 \wedge ... \wedge state_n \wedge state)$
Event-State Conditional	[*event*] should [*only*] happen when [$state_1, ..., state_n$] [*event*] should [*never*] happen when [$state_1, ..., state_n$]	$\mathbf{G}(\mathbf{X}@event \rightarrow (state_1 \wedge ... \wedge state_n))$ $\neg\mathbf{F}(state_1 \wedge ... \wedge state_n \wedge \mathbf{X}@event)$
Event-Event Conditional	[$event_1$] should [*always*] happen within [*time*] after [$event_2$] [$event_1$] should [*never*] happen within [*time*] after [$event_2$]	$\mathbf{G}(@event2 \rightarrow (time\#event2\mathbf{W}@event1))$ $\neg\mathbf{F}(time\#event_2 \wedge \mathbf{X}@event_1)$

to see four example properties (e.g., "The temperature in my bedroom should never be below 65 degrees"), while the other half did not see any examples. While both participants who did and did not see examples wrote properties following six of the seven templates, the proportion of properties matching a given template differed significantly between these two groups (χ^2, $p = .003$). Thus, we always first report the percentage among properties written by participants who did *not* see examples, followed by the percentage from those who did.

Results: We received 75 responses, discarding four who gave off-topic responses or reported having no smart devices. Of the resultant 71 participants, 64 % identified as male and 36 % as female. The median age range was 25–34 (53 %), and 9 % were age 45+. Among participants, 24 % reported a degree or job in CS or technology. Participants most frequently reported having internet-connected cameras (55 % of participants), lights (54 %), thermostats (52 %), cooking devices (18 %), door locks (15 %), and outdoor devices (8 %).

We found that seven templates captured the vast majority of desired properties participants expressed. We differentiate them based on whether they are conditional (i.e., conditioned on at least one other clause), whether they rely on a duration (i.e. expressing temporal bounds), and whether they are described based on states and/or events. The small number of remaining properties were either out of scope (e.g., requesting new features) or too ambiguous to analyze reliably.

Below are the seven templates, each with the proportion of responses that fit that template from participants who did not see examples and those who did, respectively. We also provide a sample response from participants for each template.

One-State Unconditional (40.6%, 14.7%) "Smart refrigerator should always be on."

One-Event Unconditional (24.1%, 14.5%) "My thermostat should never go above 75 degrees."

One-State Duration (0.9%, 7.5%) "My smart lights should stay on for at least 30 seconds each time."

Multi-State Unconditional (0.3%, 0.2%) "Never run the washing machine and the dish washer at the same time."

State-State Conditional (1.6%, 7.5%) "The stove should always be off if no one is home."

Event-State Conditional (26.3%, 40.7%) "My smart window should never be opened while the AC is on."

Event-Event Conditional (5.3%, 13.8%) "My smart door lock should always lock after I come in."

2.3 AutoTap property-specification interface

AutoTap aims to synthesize TAP programs satisfying user-specified properties. This section discusses our design of a property-specification user interface that aims to be expressive,

easy to use, and also compatible with LTL, allowing an easy translation from every specified property into an LTL formula.

Property types: Table 2.1 summarizes the seven property types we commonly observed in our first user study. They differ along three dimensions: whether the subject was a state or an event; whether something should or should not happen; and whether the desire was conditional or unconditional.

We note that any *state-state conditional* property can be written as an equivalent *multi-state unconditional* property. Further, some *one-state duration* properties have equivalent *event-event conditional* properties. However, to better match users' mental models, we chose not to merge these types.

Every type of property in our interface has a straightforward translation to an LTL formula, as shown in Table 2.1. The example in Figure 2.3a corresponds to a state-state conditional property: "The [*window*] should always be *closed* when [*weather*] is *raining*". It corresponds to an LTL formula $\mathbf{G}(weather.raining \rightarrow window.closed)$.

Interfaces for property specification: To not overwhelm users, AutoTap lets them first pick from three template categories, as shown in Figure 2.5, and then customize that template by selecting items from drop-down lists of devices, states, or events. Users also select whether they desire certain situation to always occur or never occur. This interface provides users with the same vocabulary about devices, states, and events as traditional TAP rule interfaces, as in Figure 2.3.

AutoTap's user interface design focuses on common user desires. It does not aim to cover all possible properties a user might think of, or all properties AutoTap synthesis can handle. As an alternative, AutoTap also allows expert users to specify safety properties directly in LTL. For example, imagine someone has a smart light bulb and wants the "red" color to always be followed by "green" or "yellow." This desire is not supported by the user interface above, yet can be described in LTL as $\mathbf{G}(color.red \rightarrow \mathbf{X}(color.green \vee color.yellow))$ and

Interface Entry	Property Type
this state and this state should always / never occur together	• Multi-state Unconditional
this state should always / never be active ⊕for this long ⊕while that	• One-State Unconditional • One-State Duration • State-State Conditional
this event should always / only / never happen ⊕while that ⊕within this long after that	• One-Event Unconditional • Event-State Conditional • Event-Event Conditional

Figure 2.5: Templates in AutoTap's property-specification UI.

thus can be handled by AutoTap.

2.4 AutoTap TAP synthesis

Problem statement: Informally speaking, smart devices continuously interact with unpredictable human users and environments. Naturally, some interactions (sequences) might cause undesirable device states or state sequences. AutoTap aims to automatically synthesize TAP programs or program patches so that all desirable situations remain intact (i.e., being *accommodating*) and all undesirable situations become disabled or transient (i.e., being *property-compliant*).

Straw-man: One potential solution is to repeatedly attempting the following two steps, as illustrated by the dashed lines in Figure 2.6: (1) propose a TAP program (patch); (2) try to prove that this program guarantees satisfaction of the desired properties, returning to Step 1 if not.

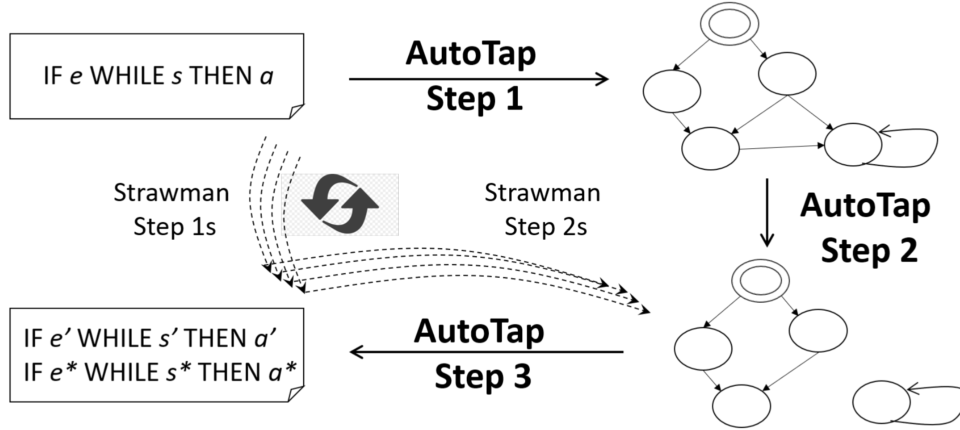

Figure 2.6: AutoTap approach vs. straw-man approach

The second step can be done through model checking [49], which typically uses a finite Büchi Automaton to represent all possible executions of the system, checking if all these executions satisfy a property ϕ by analyzing the automaton graph. Unfortunately, given the large search space of potential TAP programs, particularly when we synthesize programs from scratch, how to conduct the first step is unclear.

AutoTap approach: AutoTap takes a unique approach to solving this problem in a general and systematic way. As illustrated in Figure 2.6, it does not require iterative retries.

Step 1: Turn the given smart-device system, TAP rules (if any), and the desired property ϕ into a Büchi Automaton $\mathbb{A}$ accepting ϕ-violating executions, like what traditional model checkers do internally.

Step 2: Figure out how to modify $\mathbb{A}$ so that all ϕ-satisfying executions are kept, which guarantees being *accommodating*, and all originally accepted (i.e., ϕ-violating) executions disappear, which guarantees being *property-compliant*.

Step 3: Find *valid* TAP program(s) that can make the automaton changes suggested at Step 2.

The first step is largely straightforward, but we need to carefully model timing-related properties and avoid unnecessarily large automata. Section 2.4.1 explains how we do so.

The second step is very challenging at first glance. There are innumerable ways to change an automaton $\mathbb{A}$. It is hard to know which changes are compliant, accommodating, and

valid (e.g., changes that require modifying property ϕ and device specifications are invalid). Section 2.4.2 will present a simple algorithm that identifies such compliant, accommodating, and valid changes (i.e., a set of edges to cut in $\mathbb{A}$), leveraging a unique property of LTL safety properties. As Section 2.3 explained, the desired properties we commonly observed in our first user study all map directly to LTL safety properties.

The third step, finding valid program changes[1] that correspond to a given automaton change, is challenging for general programming languages. However, as we will explain in Section 2.4.3, it can be done in a systematic way for TAP.

2.4.1 Step 1: Model Construction

AutoTap's inputs are: (1) safety properties ϕ in LTL, obtained through the user interface presented in Section 2.3; (2) TAP rules, if any; (3) specifications for every smart device in the form of a transition system. We expect device specifications to be provided *once* by device manufacturers or tool developers like us, yet used by *all* device users. Our experiments used the specifications from Samsung SmartThings [69].

AutoTap's baseline model construction follows traditional model-checking techniques [26]. First, a transition system is built for a set of devices together with their TAP rules, if any (e.g., Figure 2.7). Some events in the transition system are controllable (e.g. "turn on the light"), while others are not (e.g. "stop raining"). This distinction is kept by AutoTap for its synthesis phase.[2] Then, this transition system is turned into a Büchi Automaton $\mathbb{A}_s$ that accepts all executions allowed in the smart-device system (e.g., Figure 2.9). Next, AutoTap applies Spot [24] to the LTL formula representing $\neg\phi$ to get a Büchi Automaton $\mathbb{A}_{\neg\phi}$ that accepts all executions violating ϕ (e.g., Figure 2.8). Finally, $\mathbb{A}_s$ and $\mathbb{A}_{\neg\phi}$ are combined into a Büchi Automaton $\mathbb{A}$ that accepts all ϕ-violating executions in the smart-device system (e.g.,

1. AutoTap does not differentiate program synthesis from patch synthesis, as the former is a special case of the latter when the original program is `null`.

2. The device specification we used [69] contains such information: capabilities with "commands" are controllable, while others can only be sensed.

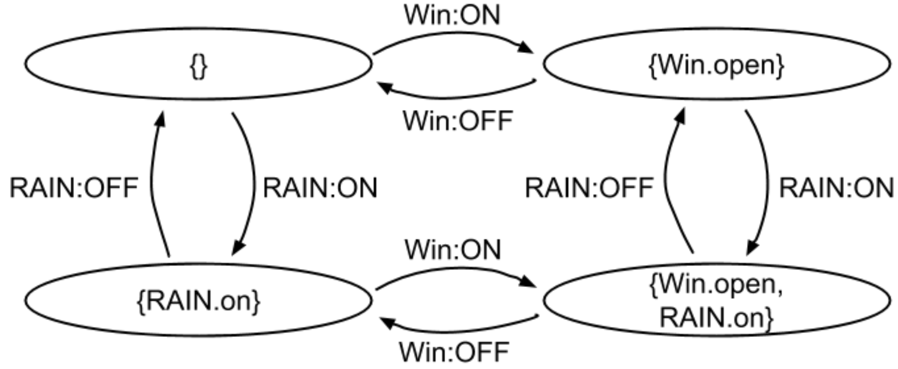

Figure 2.7: Transition system for **RAIN** and a **Win**dow. Statements in parentheses are Atomic Propositions held in each state.

Figure 2.11).

Our discussion below focuses on two techniques we developed for AutoTap beyond typical baseline modeling.

Device selection: To avoid unnecessary complexity, AutoTap selects devices D related to the given property ϕ to model. To do so, AutoTap first initializes D with all the devices that appear in ϕ. AutoTap then iteratively expands D with devices that can affect any device already in D until reaching a fixed point. Here, AutoTap considers one device to affect another device if these two both appear in a TAP rule r, with the former in the trigger and the latter in the action.

Model timing information: AutoTap extends baseline models to support timing-related propositions like "event e happened within the past t (seconds)", denoted as $t\#e$, and "ap has been true for at least t (seconds)", denoted as $t * ap$. AutoTap's property-specification interface supports both.

AutoTap first adds a count-down timer attribute timer($t\#e$) or timer(t*ap) into the transition system. The countdown starts at t, when e has just occurred, or when a system state associated with ap has just appeared. It ends at 0, indicating e has occurred or ap has been true for at least t seconds. When the system reaches a state no longer associated with ap, the $t * ap$ timer immediately flips to -1. Consequently, a state is associated with

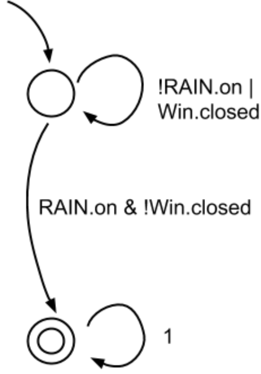

Figure 2.8: LTL property: $\neg\mathbf{G}(RAIN.on \rightarrow Win.closed)$

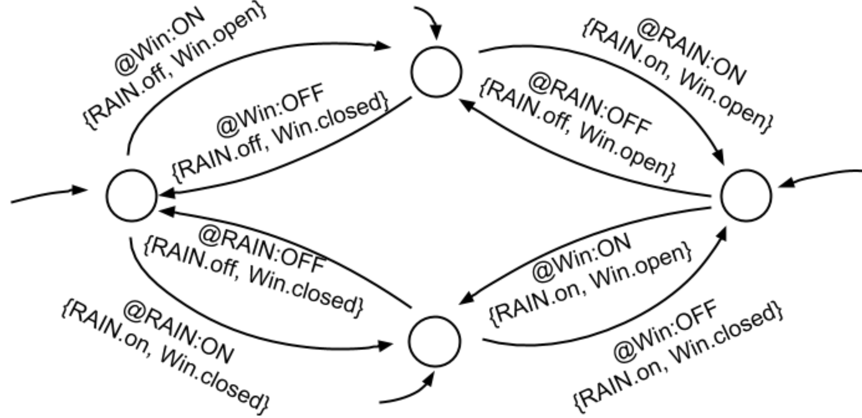

Figure 2.9: Device system: Window + RAIN

Figure 2.10: Büchi Automata of our running example.

a $t\#e$ proposition if the corresponding timer is positive. It is associated with $t{*}ap$ if the corresponding timer is 0. Then, AutoTap introduces an environmental event *tick* that counts down every positive timer uniformly. When *tick* is applied to a state s, AutoTap finds the smallest value of all the positive timers associated with s and counts down every positive timer by that value. For example, if a state is associated with three timers with values {0, 30, 100}, one *tick* will direct the system to a state with these timers being {0, 0, 70}, and another *tick* will set all three timers to 0. This count-down scheme helps AutoTap avoid unnecessary state-space explosions without losing accuracy, as counting down timers by smaller values will not change any timing related propositions (e.g., {0, 30, 100} and {0, 25, 95} will have the same set of time-related propositions).

Here, AutoTap uses its own design to handle timing-related propositions for simplicity reasons: since AutoTap only cares about two simple timed propositions $t\#e$ and $t * ap$, using more complicated timing logic like MTL [45] and more complicated timed automata [2] will only add unnecessary complexity to AutoTap property checking and rule synthesis.

2.4.2 Step 2: Patching the Automaton

The first step builds a Büchi Automaton $\mathbb{A}$ that accepts all ϕ-violating executions on smart devices. If no execution can be accepted by $\mathbb{A}$, users' desire ϕ is already guaranteed. Otherwise, this second step figures out how to change $\mathbb{A}$.

Task: We first clarify AutoTap's task at this step by reviewing some related background on Büchi Automata. By definition [26], an execution is *accepted* by a Büchi Automaton if and only if its corresponding path on the automaton visits every *accepting-node set* an *infinite* number of times. For example, the automaton in Figure 2.8 has one accepting set that consists of exactly one node, the double-circled one. It accepts every execution with a prefix ending in a state where $RAIN.on$ and $!Win.closed$ are true, which guarantees visiting the double-circled node an infinite number of times.

Consequently, AutoTap must figure out how to change $\mathbb{A}$ so that all (and only those) paths that infinitely visit $\mathbb{A}$'s accepting-node set disappear. There are several challenges. First, the change has to be *valid*, doable through possible additions or revisions of TAP rules. Naming accepting nodes as un-accepting is invalid. Deleting an edge in $\mathbb{A}$ is usually valid, as discussed in the next sub-section. Second, for arbitrary ϕ, it is difficult to tell which edges we should cut. This edge-cutting must not only eliminate every path that visits the accepting-node set infinitely (i.e., *property-compliant*), but also keeps intact every path that originally does not visit the accepting-node set infinitely (i.e., *accommodating*).

Observation: AutoTap's algorithm is based on a key observation: as long as ϕ is an LTL safety property, $\mathbb{A}$ has no edge connecting an accepting node to an un-accepting node. This observation holds because, as long as ϕ is an LTL safety property, we can always find an $\mathbb{A}_{\neg\phi}$ whose only accepting node has a single edge pointing to itself with condition 1. Once a path reaches this node, it will be stuck in this node infinitely,[3] just like the double-circled

3. Due to space constraints, we cannot include a complete formal proof. Informally, given a Büchi Automaton of an LTL safety property, all nodes corresponding to the last state of a violating prefix of the property can be replaced with an accepting node with an edge 1 pointing to itself. Those nodes can be

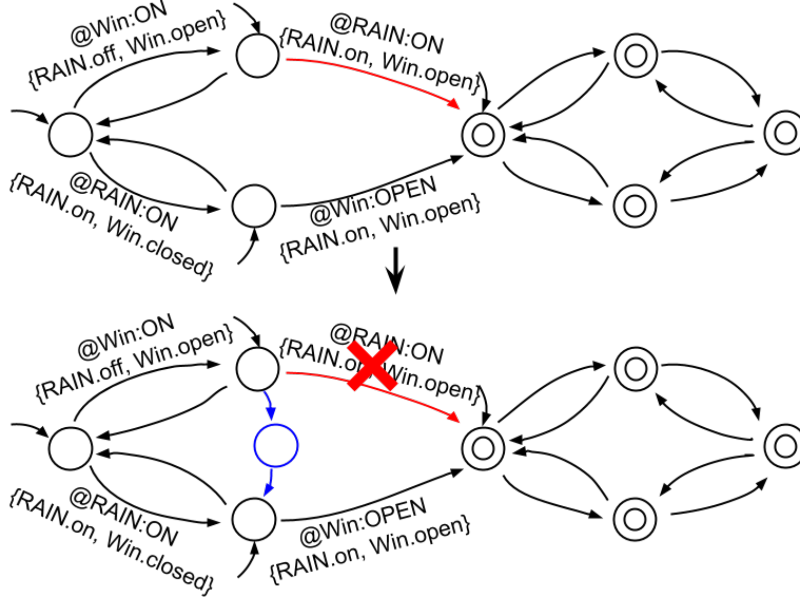

Figure 2.11: Combined Büchi Automaton of the running example. (The top is the original. The bottom is after adding a rule.)

node in Figure 2.8.

This property of $\mathbb{A}_{\neg\phi}$ then leads to the above observation of $\mathbb{A}$. The reason is that, by combining the smart-device automaton $\mathbb{A}_s$ and the property automaton $\mathbb{A}_{\neg\phi}$, every node in $\mathbb{A}$ is a cartesian product of two nodes, n_s in $\mathbb{A}_s$ and n_ϕ in $\mathbb{A}_{\neg\phi}$. The accepting-node set of $\mathbb{A}$ consists of every node whose corresponding node in $\mathbb{A}_{\neg\phi}$ is an accepting node. Furthermore, if there exists an edge from $n1$ to $n2$ in $\mathbb{A}$, there must exist an edge from $n1_{\neg\phi}$ to $n2_{\neg\phi}$ in $\mathbb{A}_{\neg\phi}$. Consequently, since there is no edge connecting the accepting node back to any un-accepting nodes in $\mathbb{A}_{\neg\phi}$, there must be no edge connecting accepting nodes back to un-accepting nodes in $\mathbb{A}$ either.

Algorithm: AutoTap identifies all the edges that connect an un-accepting node to an accepting node in $\mathbb{A}$, informally referred to as *bridge edges*, and suggests cutting all of them, like the two edges in the middle of Figure 2.11.

This algorithm is **simple**, with complexity linear in the number of edges in $\mathbb{A}$.

This algorithm is **compliant**, preventing any property violations. The reason is that, af-

combined, giving us the Büchi Automaton $\mathbb{A}_{\neg\phi}$ we desire.

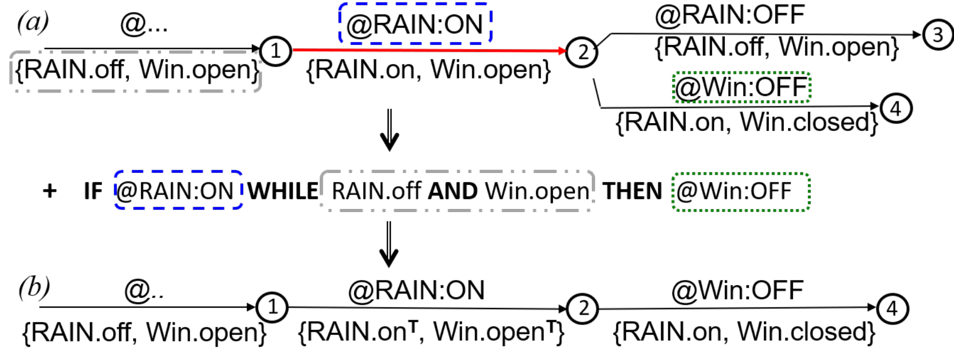

Figure 2.12: Device automaton (a) changed to (b) by adding a rule.

ter cutting all bridges, no execution can ever touch accepting nodes, not to mention infinitely. Consequently, all ϕ-violating executions are eliminated.

This algorithm is also **accommodating**, preserving all the system behaviors that do not violate ϕ. Recalling Section 2.4.2, ϕ-satisfying executions will not go through any bridges. Since our algorithm only removes or redirects bridges, yet not other edges, those executions are untouched.

2.4.3 Step 3: TAP Synthesis

At this third step, AutoTap needs to identify additions of, or revisions to, TAP rules that can delete the bridges in $\mathbb{A}$ identified in Step 2. Mapping a Büchi Automaton change to a program-code change is challenging for most imperative programming languages, but is fortunately tractable for TAP.

Task: We first clarify AutoTap's task by reviewing some background on Büchi Automata.

In $\mathbb{A}$, which is *combined* by the smart-device automaton $\mathbb{A}_s$ and the property-negation automaton $\mathbb{A}_{\neg\phi}$, every edge $e : n1 \xrightarrow{ap} n2$ is combined by an edge $e_s : n1_s \xrightarrow{ap_s} n2_s$ in $\mathbb{A}_s$ and an edge $e_{\neg\phi} : n1_{\neg\phi} \xrightarrow{ap_{\neg\phi}} n2_{\neg\phi}$ in $\mathbb{A}_{\neg\phi}$. ap is an atomic proposition (AP) set describing what is accepted by e, and e only accepts what is accepted by both e_s and $e_{\neg\phi}$. If ap_s conflicts with $ap_{\neg\phi}$, edge e would disappear from $\mathbb{A}$. To ease the discussion, we will informally refer to ap as the post-condition of $n1$ and the pre-condition of $n2$.

Since the property ϕ and the corresponding $\mathbb{A}_{\neg\phi}$ cannot be changed, AutoTap changes every bridge e's corresponding edge e_s in $\mathbb{A}_s$, which we also refer to as a *bridge*, removing e_s or changing its ap_s so that e can disappear from $\mathbb{A}$.

Example: Before presenting AutoTap's general algorithm, we use a concrete example to demonstrate how adding a TAP rule can change the smart-device automaton $\mathbb{A}_s$ and correspondingly make some edges disappear in $\mathbb{A}$.

Figure 2.12a is part of the automaton $\mathbb{A}_s$ in Figure 2.9 that models the weather (RAIN) and a smart window (Win) with no TAP rules. We can focus on node ①. Its preceding edge indicates a pre-condition when it was not raining and the window was open. Its succeeding edge ①$\xrightarrow[\{\text{RAIN.on, Win.open}\}]{@RAIN{:}ON}$② indicates that the rain starts (`@RAIN:ON`) with the post-condition being raining and window staying open. Note that this post-condition AP-set is the same as that of the bridge in $\mathbb{A}_{\neg\phi}$, illustrated in Figure 2.8. Consequently, ①→② is a bridge in $\mathbb{A}_s$ that contributes to the red bridge edge in the combined automaton $\mathbb{A}$ in Figure 2.11.

Figure 2.12b shows the effect of adding a TAP rule. As highlighted in the figure, this rule's triggering state `Rain.off AND Win.open` exactly matches the pre-condition of node ①. Its triggering event `@RAIN.ON` and rule action `@Win.OFF` exactly match the events associated with edge ① →② and edge ②→④, respectively. Consequently, immediately after ①→② takes place, this rule would automatically push the system through the ②→④ edge, essentially making the ①→② edge transient, marked by "T" in Figure 2.12. By changing the nature of ①→②, its AP-set no longer matches with that of the bridge edge in Figure 2.8. Consequently, the corresponding bridge edge in $\mathbb{A}$ (i.e., the red edge in Figure 2.11) will disappear.

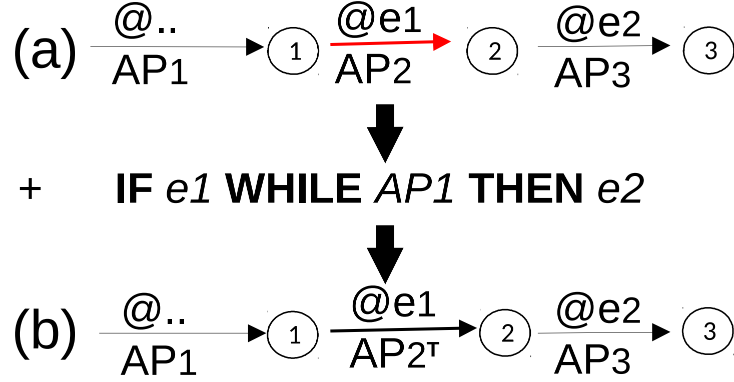

Figure 2.13: Generalization of adding TAP rules.

AutoTap fixing algorithm

We first consider a simple case where the bridge edge e_s in $\mathbb{A}_s$ has only one predecessor and one successor, as in Figure 2.13a. To cut its corresponding bridge e in the combined automaton $\mathbb{A}$, we simply need to add a TAP rule "IF e_1 WHILE AP_1 THEN e_2", where e_1 is the event associated with the bridge, AP_1 is the pre-condition of the bridge, and e_2 is the event associated with the succeeding edge. Like the example in Figure 2.12, this new rule will make states associated with e_s transient, no longer able to combine into e. That is, bridge e in $\mathbb{A}$ will be successfully cut.

Refine trigger state: The baseline algorithm uses AP_1, the bridge's pre-condition, as the trigger state of the synthesized rule. In fact, it does not have to be. We want the new rule to be triggered (1) at an original bridge edge, but (2) *not* at any non-bridge situations. The former implies that the rule's trigger-state condition should be weaker than the bridge's pre-condition. For example, since the bridge's pre-condition in Figure 2.12 is `RAIN.off AND Win.on`, the trigger state can be `RAIN.off`, or `Win.on`, or `TRUE`. The latter implies that, in other places where the trigger event could happen, the pre-conditions should conflict with the rule's trigger state, preventing the rule from being unnecessarily triggered.

To achieve this goal, AutoTap processes not only the bridge's pre-condition AP_1, but

also pre-conditions AP_i' associated with all other cases where the trigger event could occur. When there are multiple expressions satisfying the above requirements, we turn this into a hitting set problem. We use a greedy algorithm to find the smallest one.

Refine the triggered action: The baseline algorithm uses e_2 as the action of the synthesized rule because the bridge edge only has a single successor and hence e_2 is the only possible action taken in Figure 2.13. When the bridge has multiple successors with multiple possible succeeding actions, AutoTap filters out two types of actions: (1) actions that cannot be initiated by smart devices (i.e., non-controllable events like "stop raining" discussed in Section 2.4.1), and (2) actions causing other property violations. If multiple actions pass the above filtering, the only ranking AutoTap does currently is to downgrade an action that reverts the trigger event. For example, if the trigger event is turning on the air conditioner (AC), AutoTap will not suggest a rule that turns off the AC unless there are no other choices.

Revise existing rule: When the bridge edge e_s is associated with an event that is automatically triggered by an existing TAP rule r, the baseline patch would immediately trigger one TAP rule after another. A better solution is to revise r so that r is no longer triggered in this bridge situation, yet is still triggered in other situations. To achieve that, we split the general rule r into many edge-specific TAP rules by narrowing r's triggering state to only accept the pre-condition of every specific edge. Then, we simply delete the edge-specific rule associated with the bridge edge and keep the remaining ones, assuring minimum impact to the system's behavior.

Rule merging: AutoTap can merge TAP rules with the same trigger event and rule action, or even similar trigger states, to make the program easier to understand without changing system behaviors. We omit the details due to space constraints.

2.5 Evaluation

2.5.1 User Study 2: Specifying Rules vs. Specifying Properties

To evaluate usability questions regarding whether AutoTap's property-driven approach enables novice users to express their intent correctly and easily, we conducted a second online user study. In this study, we compared participants' ability to express a series of reference tasks as TAP rules (using a traditional rule-based interface) and participants' ability to express the same series of tasks as properties (using AutoTap's interface). We chose a rule-based TAP interface as our point of comparison because such interfaces are widely used [55] and prior usability studies have shown that even novice users can create TAP rules successfully [73, 34, 27, 8].

Methodology: We again recruited participants from the USA on Mechanical Turk, though for this study we did not require that they had previously used a smart device. We randomly assigned each participant to one of the following interfaces, which they used for the duration of the study:

- **Rules**: Participants created TAP rules using a web interface modeled closely after IFTTT (see Figure 2.1).
- **Properties**: Participants created properties using AutoTap's interface (see Figure 2.2)[4].

The interfaces used identical events and states. In other words, if the rule interface had an "it begins to rain" event grouped under "weather," so did the property interface.

Participants began the study by completing a short tutorial on their assigned interface. The tutorial explained key concepts (e.g., the difference between events and states) and included attention-check questions. These questions automatically pointed out the right

4. At the time of the study, our interface let users specify positive Event-State Conditional properties through an "event E should always happen while state S is true" template. Afterwards, we replaced "always" with "only" to avoid ambiguity, as shown in Table 2.1 and Figure 2.5. For participant answers using this "always" template, we interpret them as "E should be triggered while S becomes true," in this way judging three participants' answers to be correct.

answer for anything participants answered incorrectly. We designed the two tutorials to have parallel structure and share examples as much as possible.

Participants then used their assigned interface to complete 7 tasks randomly selected (and randomly ordered) from a larger set of 14. We developed each of the 14 tasks based on desired properties expressed in Study 1. However, we rewrote the tasks so that the wording of the task would not make obvious which property template should be used. An example task follows:

> You have a Roomba robotic vacuum cleaner in your home, and you've given it a schedule for when it should clean the floor. However, when the curtains in your home are open, the drawstring lays on the floor and often causes the Roomba to get stuck on the string. You want to make sure this does not happen again.

This task could be completed successfully with the rules "IF *Roomba becomes on* WHILE *the curtain is open*, THEN *close the curtain*; IF *curtain becomes open* WHILE *Roomba is on*, THEN *turn off Roomba*" or the property "*Roomba is on* should *NEVER* be active WHILE *curtain is open*". We constructed the set of tasks so that at least two tasks could be completed with each of the 7 property templates. Since many properties can be expressed in multiple ways, though, most templates could be used for more than two tasks.

After each task, participants rated their confidence in their submission and perception of how difficult it was to complete the task on five-point scales. They also had the opportunity to explain, in free text, any corner cases they had considered. After completing all 7 tasks, they filled out demographics questions and the standardized System Usability Scale.

We analyzed our data as follows. Since many tasks could be completed in multiple ways, two researchers independently coded each response as "correct," "partially correct," or "completely incorrect," meeting to resolve discrepancies. The "partially correct" category was used when a response did not address a corner case. To compare categorical data (e.g., the distribution of correct/incorrect responses), we used the χ^2 test. To compare ordinal

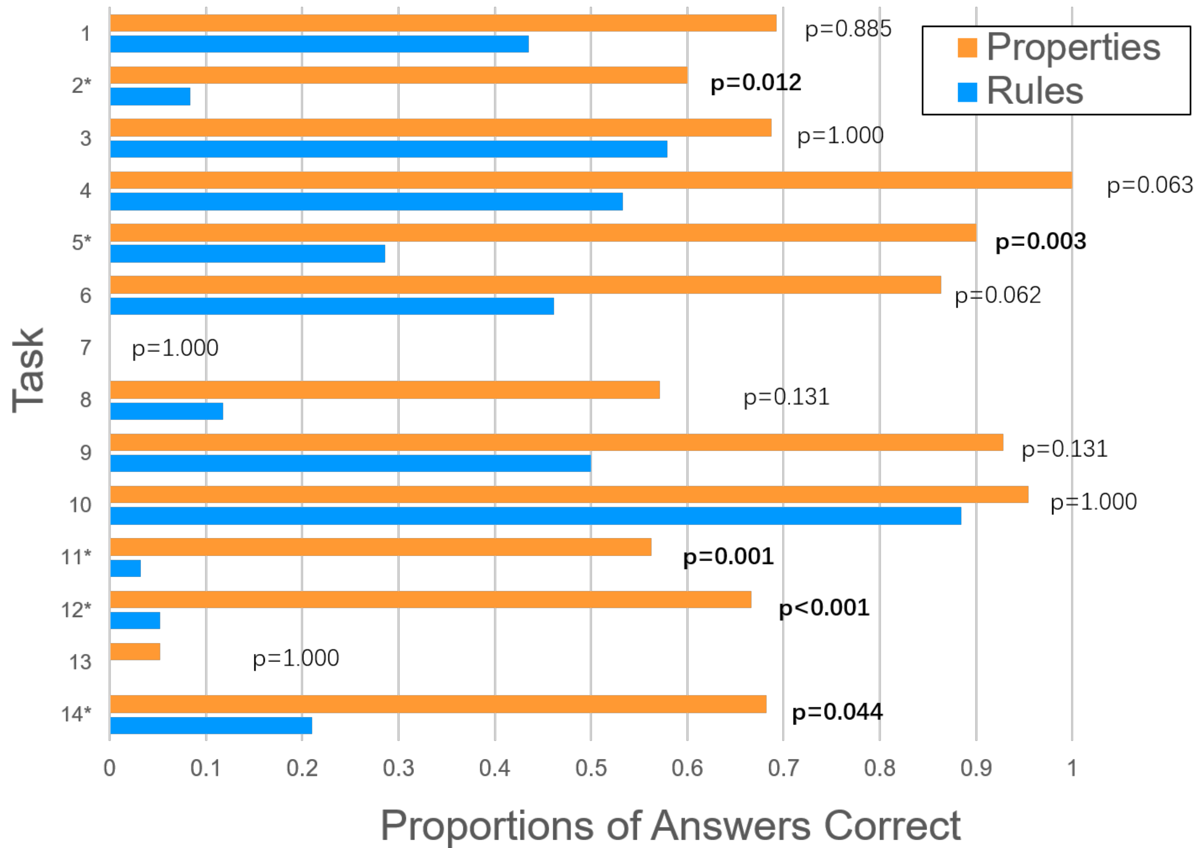

Figure 2.14: Correctness of properties and rules by task. P-values are from Holm-corrected χ^2 tests comparing the proportion of statements correct when written using rules versus properties.

data (e.g., confidence) we used the Mann-Whitney U test. To correct p-values for multiple testing, we used the Holm method within each family of tests.

A key limitation is that the 14 tasks were not intended to be a representative sample of all desired behaviors in TAP systems. Because the tasks were based in part on Study 1, they likely over-represent behaviors that can be expressed as properties. While our study can show whether some tasks are easier to express as rules or safety properties, the proportion of tasks for which this is the case is not generalizable.

Results: A total of 81 Mechanical Turk workers participated in Study 2. Three gave nonsensical free-response answers, leaving 78 valid participants.

For all 14 tasks, the percentage of correct responses was higher for AutoTap's property-

creation interface than for the TAP rule interface. This difference was statistically significant for five of these tasks (the bolded p-values in Figure 2.14). The tasks for which we observed significant differences generally required multiple rules to capture all corner cases. For example, in the aforementioned Roomba task (Task 11 in Figure 2.14), only one property is needed: "*the window curtains are open* SHOULD NEVER BE ACTIVE WHILE *the Roomba is on*." AutoTap automatically generates rules to satisfy this property in all situations. However, two rules are required. One possibility is a rule closing the curtains whenever the Roomba turns on, and another turning off the Roomba whenever the curtain is opened. Under 5% of participants wrote both of these rules. While over 55% of participants who used the property interface solved this task, one particular error appeared commonly. The property "*the curtain is open* AND *the Roomba is on* SHOULD ALWAYS OCCUR TOGETHER" inadvertently binds the two states, causing the Roomba to start anytime the curtain is opened, misinterpreting the intent.

Participants often performed similarly with the rule and property interfaces when both a single rule and a single property sufficed. For example, Task 3 (preventing a room from getting too hot) required only one of each. Participants performed similarly with either interface. AutoTap's property interface was more successful when multiple rules were needed to capture corner cases. Two tasks caused participants great difficulty, even for properties. Task 7 required either two properties or six rules. All participants missed corner cases. Task 13 dealt with delaying vacuuming when guests were over, requiring either two properties or two rules. Most participants neglected to start the vacuuming after a delay.

We compared the System Usability Scale scores provided by users to the rule interface and AutoTap property interface. We found both interfaces to be "usable", with mean scores of 70.4 and 63.2 respectively. This difference was not statistically significant (Mann-Whitney $U = 590.5$, $p = .052$).

2.5.2 TAP Program Synthesis

We further check if AutoTap can synthesize TAP rules from scratch to accomplish all 14 tasks in this user study. In a less challenging version, one of the authors (representing an expert user) wrote properties for every task, and AutoTap successfully synthesized TAP rules for all tasks.

In a more challenging version, we used *all* the correct properties written by user-study participants (158 sets of properties in total, with each from one participant targeting one task). Sets contain 1.83 properties on average. These properties were transformed into LTL formulas following Table 2.1. AutoTap successfully generated TAP programs for 157 out of the 158 property sets, and all are *guaranteed* to satisfy corresponding properties. The only set that AutoTap failed to synthesize is for "When Bobbie is in the kitchen, the oven door should be closed" and "When Bobbie is in the kitchen, the oven door should be locked." If Bobbie enters the kitchen when the oven door is open, the system needs to trigger *two* actions immediately, both closing and locking the oven door. AutoTap fails to find a solution because it currently only considers using a single action to redirect each bridge edge in the Büchi Automaton. Future work can extend AutoTap to consider using multiple actions to redirect a bridge, addressing this limitation.

We also checked how many TAP program candidates AutoTap generates for one property set. On average, AutoTap generates 2.13 candidates for one set, with a median of 1. The largest set contains 27 candidates. This is a special case as the program consists of three rules. For every rule, the potential action could be opening any one of three windows in a house. Even in this case, end users will not face 27 candidates at once. They will only need to make a one-out-of-three choice three times. As all candidates satisfy users' desires, AutoTap can also randomly pick one candidate.

Table 2.2: How AutoTap fixes buggy TAP programs. Subscripts are the # of cases AutoTap patches revert the mutation.

Source	#buggy TAP sets	Successful Fixing
mutation: change trigger event	5	4_1
mutation: add condition	7	7_7
mutation: change condition	5	5_1
mutation: change action	4	3_0
mutation: delete rule	4	4_4
Total	25	23_{13}

2.5.3 TAP Program Fixing

We randomly take 10 correct TAP program written by user-study participants and apply a wide variety of mutations to them, as shown in Table 2.2. AutoTap successfully fixes the buggy TAP program to satisfy the given property in 23 out of 25 cases, showing its generality across different types of TAP bugs. The two cases where AutoTap fails are like the following. The task is "the thermostat should never be above 80°F", and the rule is "IF *thermostat goes above* 80°F, THEN *set thermostat to* 81°F", with the action randomly mutated from "*set thermostat to* 75°F". Since the buggy rule triggers itself *recursively* and AutoTap does not regard intermediate triggering states as violating properties, AutoTap could not identify the bridge edges and hence did not repair the program.

As also shown in Table 2.2, AutoTap often generates a patch to revert the add-condition mutation or the delete-rule mutation, but not for all types of mutations. The reason is that AutoTap only fixes the part of a TAP program that violates the safety property. If a rule becomes a non-violating different rule after mutation, AutoTap will not revert the mutation back.

2.5.4 Handling Multiple Properties

Properties that share the same capabilities of devices sometimes interfere with each other. We evaluated AutoTap on 7 scenarios where such things happened, with each scenario combining

different property sets in our user study together. For example, one scenario could contain two properties "the living room window, the bedroom window and the bathroom window should never be closed together (ϕ)" and "the living room window should always be closed while it is raining (ψ)".

AutoTap simply combines different properties ϕ and ψ together as $\phi \wedge \psi$. It successfully handles all scenarios by generating TAP programs to satisfy every multi-property scenario unless the properties conflict with each other. In the latter case, AutoTap correctly reports that no TAP rules can possibly guarantee all the properties. One example of conflicting properties is "the window should always be open" and "the window should never be open when the air conditioner is on."

2.6 Conclusions

With the wide adoption of smart devices, helping users correctly express their intent for how these devices should interact is crucial. AutoTap helps users by allowing them to directly specify properties they wish to hold, rather than writing rules for exactly how devices should behave in order to satisfy those properties. To achieve this goal, we first conducted a user study to map the properties users commonly desire. We then designed an easy-to-use interface for property specification and a technique supported by formal methods to automatically synthesize TAP programs or program patches that guarantee the system satisfies the specified properties.

CHAPTER 3

TRACE2TAP: SYNTHESIZING TRIGGER-ACTION PROGRAMS FROM TRACES OF BEHAVIOR

3.1 Introduction

Consumer Internet of Things (IoT) smart devices have become very common [39]. With automation, these smart devices can react to environmental contexts and user behaviors, potentially improving users' quality of life by streamlining their daily routines [1, 70]. 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)** [73, 33, 74, 27], 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 [7, 33, 52] or struggle to understand why particular automations are running [54, 79, 53]. In short, while prior work has shown that even non-technical users can write simple trigger-action programs with ease [73], 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 [67, 4], 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 3.4.1). It then applies symbolic execution and SAT-solving techniques [18] 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 3.4.2). Our novel use of symbolic reason-

ing 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 3.5.1). 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 3.5.2).

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 3.4.3). The rule-presentation interface mentioned above then helps the user understand the impact of each patch and make a proper debugging decision.

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.

3.2 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.

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

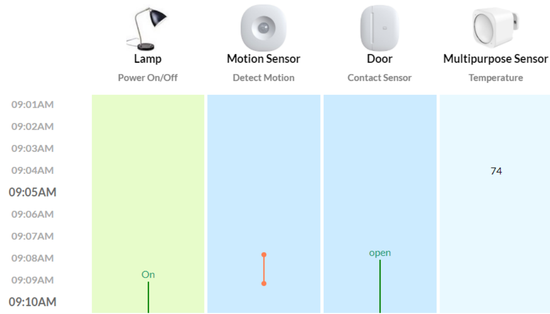

Figure 3.1: User comes to work.

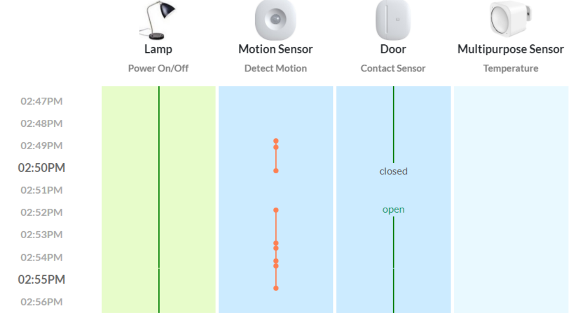

Figure 3.2: User temporarily leaves the office.

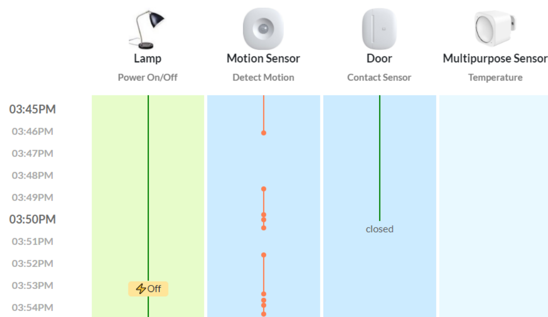

Figure 3.3: User closes the door for work.

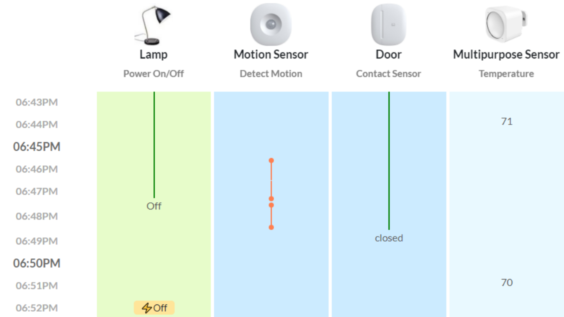

Figure 3.4: User goes home.

Figure 3.5: Visualizations of different use cases in the trace collected from an example office occupant.

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 3.5), 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

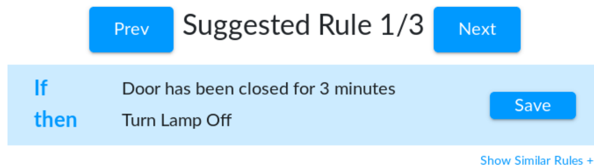

Figure 3.6: Rule 1 in Cluster 1 for turning off the lamp.

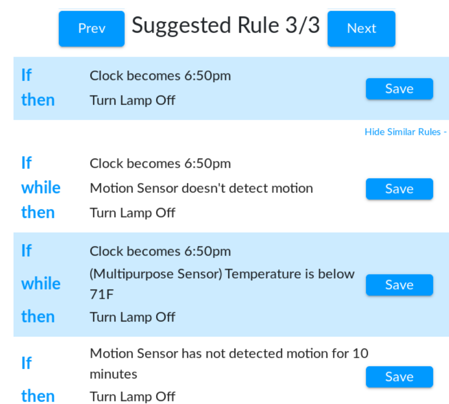

Figure 3.7: All rules in Cluster 3, shown on demand.

Figure 3.8: An example of Trace2TAP's UI for showing proposed rules to the occupant.

them. While in their office, they usually left the door open. However, they also sometimes closed their door to concentrate.

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 3.8. The UI also visualized relevant sensors and devices from the trace, as in Figure 3.5. Upon demand, the user could show all rules in the cluster, as in Figure 3.7. 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 3.8) 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 3.9) 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.

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 4.5), Trace2TAP's algorithm handles events that appear out of order. Because of the physical location of the

Prev Suggested Change 1/10 Next Apply

If Door has been closed for 3 minutes
while + Motion Sensor doesn't detect motion
then Turn Lamp off

Stats and Visualization

You reverted the automation "Turn Lamp off" 2 times.
The change would cancel the automation 1 time when you actually reverted them.
In some cases when you didn't revert the action,
the change would also cancel 0 automated actions.

Figure 3.9: The patch Trace2TAP suggested to modify Rule 1 from Cluster 1 based on the user reverting automations.

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 3.4) 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.

3.3 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 [58] 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**" [7]. 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 [32, 58], and prior work has found that it is the least ambiguous TAP format [7, 63].

As we will explain in Section 3.4.2, 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.

3.4 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 3.4.1. The second step is the key that allows Trace2TAP to be *inclusive*. It will be explained in Section 3.4.2. 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 3.4.3.

3.4.1 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.

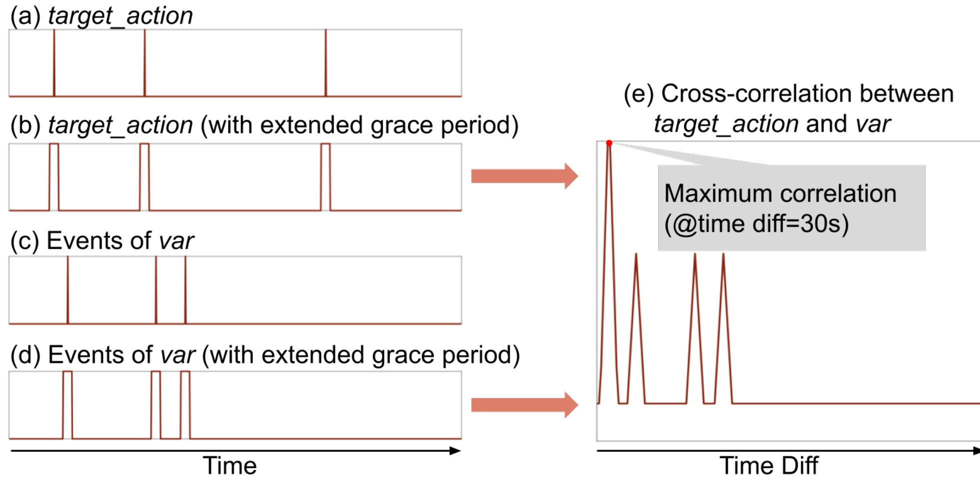

Figure 3.10: Calculating how a variable *var* is related to *target_action*.

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 each pulse lasting for a certain time period (15 minutes by default in our current implementation), as illustrated in Figure 3.10. 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 3.4.2).

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 [7], 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

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.

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 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}.

3.4.2 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 [18] to handle the last task. Here, we focus on Step 1 and Step 2.

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 3.1, 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,

Table 3.1: 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), "$\neq$" (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 "$V_{t_1} \otimes_{t_1} K_{t_1}$".

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 3.1, 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 3.1 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`; $\oplus_{V_{c_3}} := \;\neq\;$; $\boldsymbol{K}_{V_{c_3}} :=$ `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 $\mathbb{A}$.

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 $var : R.\oplus_V =$ " $=$ " $\wedge R.\otimes_V =$ " $=$ ": "= 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:

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

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 3.4.2. The constraint above describes that at the end of the *episode*, the effect of action $\mathbb{A}$ has to be there: $S'[A.var] == A.val$.

3. Δ_1 and Δ_2 are configurable. In our current prototype, they are set to 10 minutes and 5 minutes, respectively.

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:

$$\sum_{i=1}^{n}[S'_i[A.var] == A.val] \geq threshold \times n \tag{3.1}$$

S'_i = `executeTrace` $(episode_i, R_1 \ldots R_m, R)$, and "$[\cdots]$" is the indicator function of a Boolean expression.

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

4. In our current study, we set *threshold* to 0.3.

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.

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 3.6, 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 ... 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 ... R_j do
                triggered := checkRule(r, E, S);
                if triggered then
                    S[r.action.var] := r.action.val;
                end
            end
        end
    end
    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**
  - $trig := \big((E.val)R. \otimes_{t_i} (R.\boldsymbol{K}_{t_i})\big) \land R.\boldsymbol{\lambda}_{t_i}$;
- **else**
  - $trig :=$ false;
- **end**
- **for** $i := 1 \dots 3$ **do**
  - $cond_i := R.\boldsymbol{\mu}_{c_i} \rightarrow \big((S[V_{c_i}])R. \oplus_{c_i} (R.\boldsymbol{K}_{c_i})\big)$;
- **end**
- $cond := cond_1 \land \cdots \land cond_n$;
- $result := trig \land cond$;
- **return** *result*;

Algorithm 2: Check if symbolic rule R of the format in Table 3.1 is triggered. We use $R.\boldsymbol{K}$, $R.\otimes$, and so on to refer to symbols in R.

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 3.1) 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 3.4.1, 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 3.1 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.

3.4.3 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.

IF **trigger** WHILE **conditions** AND $\begin{matrix} \boldsymbol{\mu}_{V_{c_1}} \to V_{c_1} \oplus_{c_1} \boldsymbol{K}_{c_1} \\ \boldsymbol{\mu}_{V_{c_2}} \to V_{c_2} \oplus_{c_2} \boldsymbol{K}_{c_2} \\ \dots \\ \boldsymbol{\mu}_{V_{c_m}} \to V_{c_m} \oplus_{c_m} \boldsymbol{K}_{c_m} \end{matrix}$ 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:

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

Then, similar to Equation 3.1, 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 ($> threshold_2$).

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

3.5 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 3.5.1, we discuss how Trace2TAP clusters and ranks rules. In Section 3.5.2, we then show how Trace2TAP visualizes each rule's rationale and implications.

3.5.1 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.

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.

Trace2TAP then automatically clusters all rule candidates based on the Hamming distance between their bit vectors, using the K-modes [35] 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 [68], limiting the maximum number of clusters to five.

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 3.11).

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

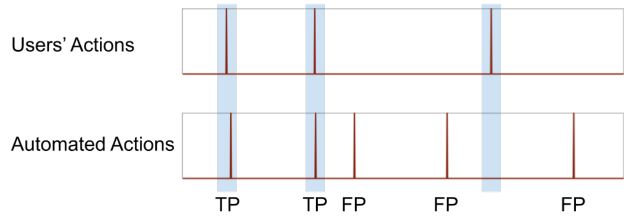

Figure 3.11: Counting a rule's true positives and false positives.

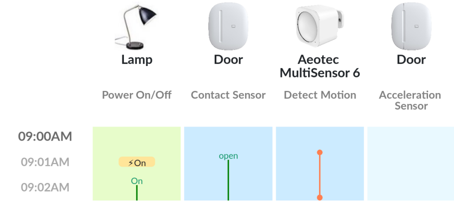

Figure 3.12: A visualization of when in the trace the rule would have triggered (the "on" with the beige background).

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 3.8). 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.

3.5.2 *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 3.6 from Section 3.2 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 3.12, 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 3.12'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.

3.6 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 3.1, 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 [75]; (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 3.2, 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 [38], albeit with our expanded TAP syntax (Section 3.3).

3.7 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 3.2. 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]

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 3.2).

3.8 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 3.8.1 and 3.8.3), how important it is for rule synthesis to be *comprehensive* (Section 3.8.2), and what factors influenced participants' decision processes (Section 3.8.4).

3.8.1 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 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

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.

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 3.13, 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 3.14. 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 3.14 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

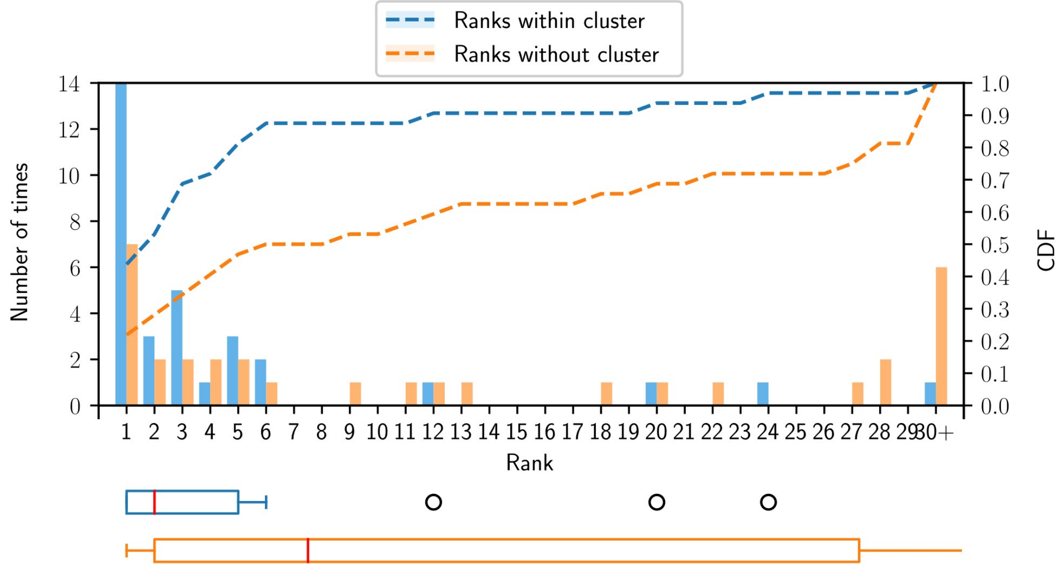

Figure 3.13: Rank distribution of rules selected by participants with and without clustering. The dashed curves are CDFs.

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.

3.8.2 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 3.15. 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

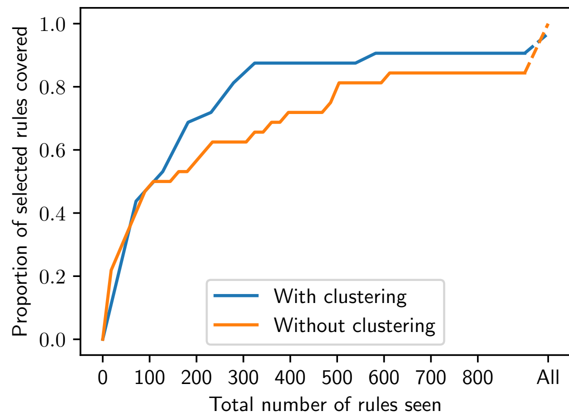

Figure 3.14: Coverage of selected rules by the number seen. The x-axis is aggregated from rules for all target actions.

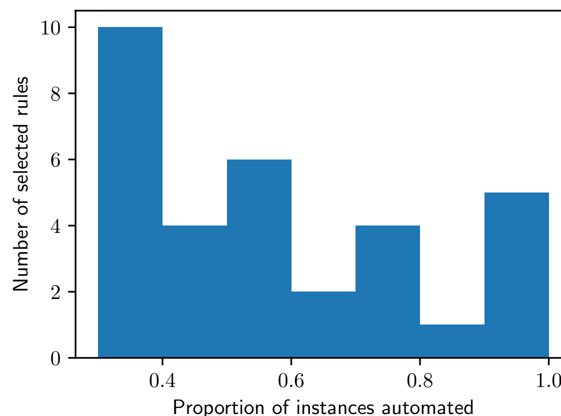

Figure 3.15: The proportion of manual instances automated.

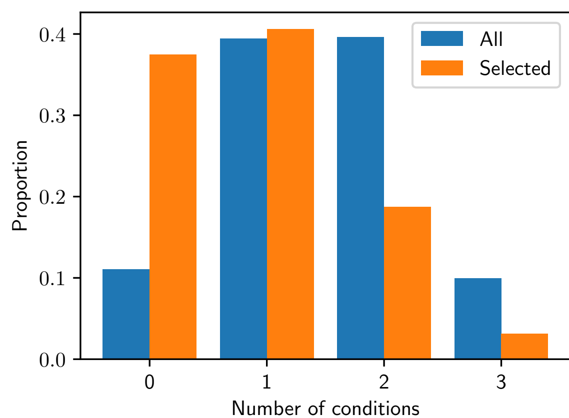

Figure 3.16: The number of conditions rules selected by participants have, compared to all rules synthesized.

Figure 3.17: The distribution of the ranks of clusters with at least one rule selected.

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 3.16, 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

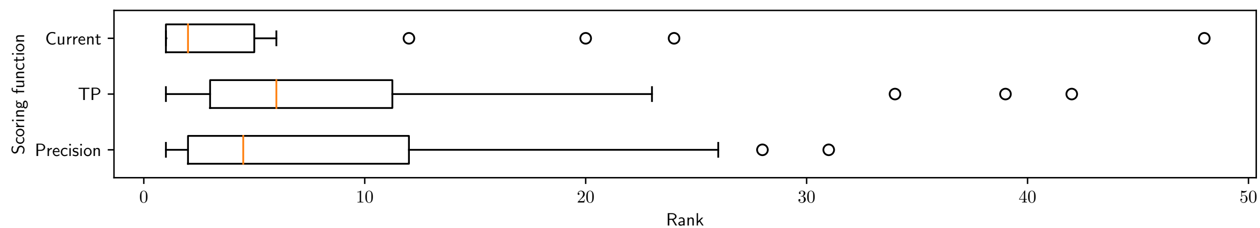

Figure 3.18: The ranks within a cluster of rules selected by participants under the current scoring function, as well as under potential alternatives relying on true positives (TP) and precision.

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.

3.8.3 How Effective is the Ranking Function?

Trace2TAP uses its ranking function, which was discussed in Section 3.5.1, 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 3.18, 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 participant in the field study to at least look at every cluster. Figure 3.17 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.

3.8.4 Qualitative Analysis of Participants' Rule-Selection Processes

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

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 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

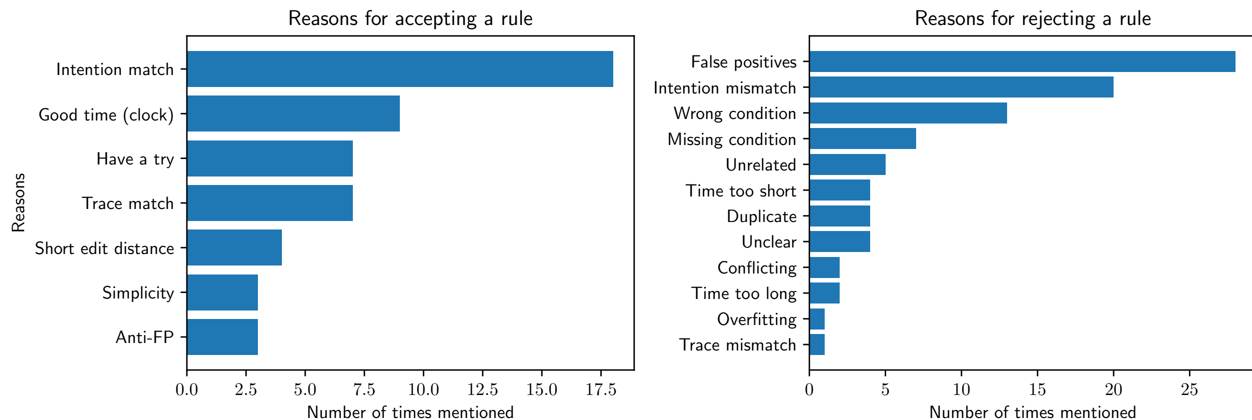

Figure 3.19: The number of times participants stated the given reason for accepting/rejecting a rule in our qualitative interviews.

then met to resolve disagreements. The result is shown in Figure 3.19. 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.

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.

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 3.20 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 3.4).

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 [67] 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 3.21 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

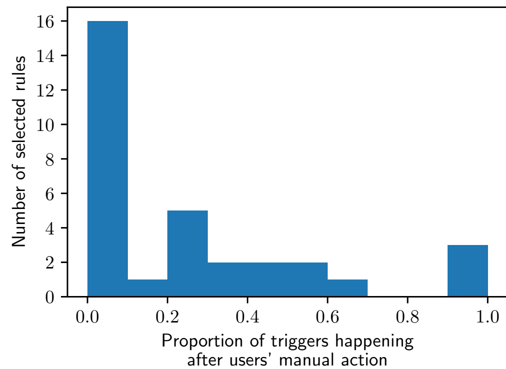

Figure 3.20: Number of selected rules that do not follow event timing in the traces (timing mis-order).

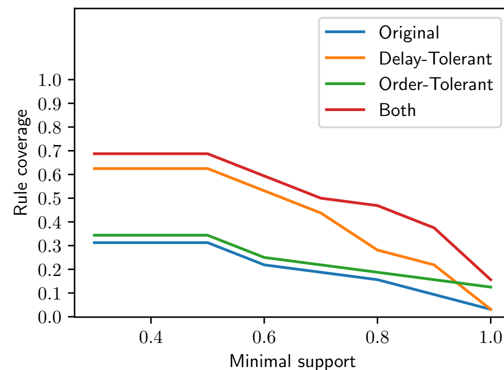

Figure 3.21: The coverage of selected rules using variants of prior work's pattern mining approach, which is parameterized by the minimal support.

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 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.

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.

3.9 Conclusion

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 [5, 23, 25, 78, 54], 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").

CHAPTER 4

RELATED WORK

4.1 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 [74, 55] systems including IFTTT [38], Zapier [44], Microsoft Flow [48], Samsung SmartThings [58], Mozilla Things Gateway [36], OpenHAB [62], and Home Assistant [32]. Initial user studies found that non-technical users could accurately write TAP rules to automate smart spaces in simple scenarios [73, 77, 59, 27]. However, more recent work has observed shortcomings of TAP in more complex scenarios [8, 15], in particular relating to users writing rules that contain bugs or otherwise fail to match their intent [7, 79, 33]. Furthermore, users often find it hard to reason about how sensors (e.g., motion sensors) in smart homes work [31]. For example, they may misunderstand what brightness-level readings mean.

Tools have been developed to detect some bugs in TAP programs [75, 72, 20, 16]. 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 [60]. Others leverage constraint solving and/or model checking techniques to automatically synthesize TAP rules or rule patches that satisfy specific constraints [49, 9, 80]. 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.

4.2 TAP program bug-detection and fixing

AutoTap is inspired by previous work [49, 9] that applied formal methods to detect violations to LTL or CTL policies in TAP programs. Previous work searches potential TAP patches by changing trigger-states of *existing* TAP rules in three ways: (1) deleting a conjunction clause; (2) adding a conjunction clause that appears in the LTL/CTL policy; or (3) modifying numerical parameters. Consequently, they cannot synthesize patches that change TAP rules' trigger events or rule actions, not to mention creating new TAP rules from scratch. The end-user property-specification interface of previous work [9] only accepts "[*states*] shall not happen", missing many common desires.

TrigGen [60] detects a specific type of bug in OpenHAB TAP programs [62] – missing triggers. It works by checking what events not included in the trigger could possibly affect the rule conditions. Researchers have also developed techniques for either crowdsourcing TAP rules [34] or synthesizing TAP rules from natural language [13, 66]. Our synthesis and repair techniques are complementary to those techniques.

4.3 Program synthesis using formal methods

Synthesizing a program from a formal specification, or *LTL synthesis*, has been an open problem [6]. Most work in this area synthesizes reactive systems based on formal specifications [10, 6, 64, 47]. AutoTap is related to, but also fundamentally different from, such work. AutoTap needs to synthesize TAP rules, not just finite state models, and needs to accommodate for an existing finite state model (i.e., the smart-device system model). Degiovanni et

al. proposed an algorithm that synthesizes control-operation programs, which have similar syntax as TAPs, to satisfy formal requirements [21]. Due to the different usage contexts, their algorithm, which uses SAT solvers to iteratively resolve counter-examples by changing existing rules' trigger states, cannot add new rules or preserve existing property-compliant behaviors.

4.4 Property-specification interfaces

Past work in requirements engineering investigated how to let engineers specify desired software properties. KAOS provided guidelines that helped engineers gradually summarize or break down vague requirements into deployable specifications [17]. PSPWizard provided an interface where developers could choose from a comprehensive list of templates, fill in the blanks of the chosen template, and then have their inputs translated into formal specifications [51]. In contrast with those efforts, we employed a user study to identify commonly desired properties in smart-home scenarios. We then designed property-specification templates for expressing those properties through a compact graphical interface. AutoTap users specify properties through only mouse clicks, which is suitable for non-technical users.

4.5 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 [67]. 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 [56, 57]. 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 [4] 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 3.8) 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.

4.6 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 [18], 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 "Satisfia-

bility Modulo Theories (SMT)" problem. Although even an SMT sub-problem, the Boolean satisfiability problem (SAT), is NP-complete [28], state-of-the-art SMT solvers deploy smart searching strategies [19] to solve typical SMT problems efficiently. They are widely used in static program analysis, hardware/software verification, and test-case generation [19].

SAT/SMT solvers are efficient tools to explore the search space in program synthesis [29]. Previous work has synthesized various types of programs using logical specifications [71, 40] or input-output examples [42] 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 [42], regular expressions [61], or more [29]. 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.

4.7 Context-Aware Computing

Context-aware systems customize services based on the context of usage [22]. 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 [46, 65], provide personalized services [82], and even make crucial decisions regarding security and privacy [3, 43]. 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 [5, 23, 25, 78, 54]. 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 [50, 41, 37] and designed interfaces that establish constructive collaboration between systems and users [14, 53, 30].

CHAPTER 5

SUMMARY

This proposal introduces our efforts to enhance end-users' trigger-action programming experience. We introduced AutoTap, a system that lets novice users specify desired properties for devices and services. It translates these properties to linear temporal logic (LTL) and synthesizes property-satisfying TAP rules [80]. We also designed Trace2TAP, which synthesizes TAP rules from users' past behaviors and presents them to end-users in a carefully designed way [81]. Finally, we proposed TapDebug, a system that helps end-users at different stages in their TAP debugging process with automated tools.

REFERENCES

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

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

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

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

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

[6] Rastislav Bodik and Barbara Jobstmann. Algorithmic program synthesis: Introduction. *International Journal on Software Tools for Technology Transfer*, 15(5):397–411, Oct 2013.

[7] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. How users interpret bugs in trigger-action programming. In *Proc. CHI*, 2019.

[8] Julia Brich, Marcel Walch, Michael Rietzler, Michael Weber, and Florian Schaub. Exploring end user programming needs in home automation. *ACM TOCHI*, 24(2):11, 2017.

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

[10] J Richard Büchi and Lawrence H Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[11] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. SOTERIA: Automated IoT safety and security analysis. In *Proc. USENIX ATC*, 2018.

[12] Ryan Chard, Kyle Chard, Jason Alt, Dilworth Y. Parkinson, Steve Tuecke, and Ian Foster. Ripple: Home automation for research data management. In *Proc. ICDCSW*, 2017.

[13] Xinyun Chen, Chang Liu, Richard Shin, Dawn Song, and Mingcheng Chen. Latent attention for if-then program synthesis. In *Proc. NIPS*, 2016.

[14] 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. 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*, 2020.

[15] Meghan Clark, Mark W. Newman, and Prabal Dutta. 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):44, 2017.

[16] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. Empowering end users in debugging trigger-action rules. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2019.

[17] Robert Darimont, Emmanuelle Delor, Philippe Massonet, and Axel van Lamsweerde.

GRAIL/KAOS: An environment for goal-driven requirements engineering. In *Proc. ICSE*, 1997.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proc. TACAS*, 2008.

[19] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[20] Luigi De Russis and Alberto Monge Roffarello. A debugging approach for trigger-action programming. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018.

[21] Renzo Degiovanni, Dalal Alrajeh, Nazareno Aguirre, and Sebastian Uchitel. Automated goal operationalisation based on interpolation and SAT solving. In *Proc. ICSE*, 2014.

[22] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[23] Anind K. Dey and Alan Newberger. Support for context-aware intelligibility and control. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009.

[24] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0—a framework for ltl and ω-automata manipulation. In *Proc. ATVA*, 2016.

[25] W. Keith Edwards and Rebecca E. Grinter. At home with ubiquitous computing: Seven challenges. In *Proceedings of Ubicomp*, 2001.

[26] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. PSTV*. Springer, 1995.

[27] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. Personalization of context-dependent applications through trigger-action rules. *ACM TOCHI*, 24(2):14, 2017.

[28] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: A survey. Technical report, Cincinnati University Department of Electrical and Computer Engineering, 1996.

[29] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2010.

[30] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home internet of things (iot). In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[31] Weijia He, Jesse Martinez, Roshni Padhi, Lefan Zhang, and Blase Ur. When smart devices are stupid: Negative experiences using home smart devices. In *Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW)*, 2019.

[32] Home Assistant. https://www.home-assistant.io/docs/automation/.

[33] Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In *Proc. UbiComp*, 2015.

[34] Ting-Hao Kenneth Huang, Amos Azaria, and Jeffrey P Bigham. Instructablecrowd: Creating if-then rules via conversations with the crowd. In *Proc. CHI Extended Abstracts*, 2016.

[35] Zhexue Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data Mining and Knowledge Discovery*, 2(3):283–304, 1998.

[36] Matthew Hughes. Mozilla's new Things Gateway is an open home for your smart devices. TheNextWeb, February 7, 2018.

[37] Jan Humble, Andy Crabtree, Terry Hemmings, Karl-Petter Åkesson, Boriana Koleva, Tom Rodden, and Pär Hansson. "playing with the bits" User-configuration of ubiquitous domestic environments. In *Proceedings of Ubicomp*, 2003.

[38] IFTTT. https://ifttt.com, 2020.

[39] Insider Intelligence. How IoT devices & smart home automation is entering our homes in 2020. Business Insider, Jan 6, 2020. https://www.businessinsider.com/iot-smart-home-automation.

[40] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. *ACM Sigplan Notices*, 45(10):36–46, 2010.

[41] Timo Jakobi, Gunnar Stevens, Nico Castelli, Corinna Ogonowski, Florian Schaub, Nils Vindice, Dave W. Randall, Peter Tolmie, and Volker Wulf. 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):171:1–171:28, 2018.

[42] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd International Conference on Software Engineering*, 2010.

[43] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, and Atul Prakash. Contexlot: Towards providing contextual integrity to appified iot platforms. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, 2017.

[44] Thorin Klosowski. Automation showdown: IFTTT vs Zapier vs Microsoft Flow. LifeHacker, June 26, 2016.

[45] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[46] Joohyun Lee, Kyunghan Lee, Euijin Jeong, Jaemin Jo, and Ness B. Shroff. 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*, 2016.

[47] Emmanuel Letier and William Heaven. Requirements modelling by synthesis of deontic input-output automata. In *Proc. ICSE*, 2013.

[48] Nat Levy. Microsoft updates ifttt competitor flow and custom app building tool powerapps. GeekWire, April 17, 2017.

[49] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. Systematically debugging IoT control system correctness for building automation. In *Proc. BuildSys*, 2016.

[50] Brian Y. Lim and Anind K. Dey. Assessing demand for intelligibility in context-aware applications. In *Proceedings of the 11th International Conference on Ubiquitous Computing*, 2009.

[51] Markus Lumpe, Indika Meedeniya, and Lars Grunske. PSPWizard: machine-assisted definition of temporal logical properties with specification patterns. In *Proc. ESEC/FSE*, 2011.

[52] Marco Manca, Fabio Paternò, Carmen Santoro, and Luca Corcella. Supporting end-user debugging of trigger-action rules for IoT applications. *International Journal of Human-Computer Studies*, 123:56–69, 2019.

[53] Sarah Mennicken, David Kim, and Elaine May Huang. Integrating the smart home into the digital calendar. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2016.

[54] Sarah Mennicken, Jo Vermeulen, and Elaine M. Huang. 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*, 2014.

[55] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. An empirical characterization of IFTTT: Ecosystem, usage, and performance. In *Proc. IMC*, 2017.

[56] Bryan Minor, Janardhan Rao Doppa, and Diane J. Cook. Data-driven activity prediction: Algorithms, evaluation methodology, and applications. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 805–814, 2015.

[57] Bryan David Minor, Janardhan Rao Doppa, and Diane J. Cook. Learning activity predictors from sensor data: Algorithms, evaluation, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2744–2757, 2017.

[58] Walt Mossberg. SmartThings automates your house via sensors, app. Recode.net, 2014.

[59] Alessandro A. Nacci, Bharathan Balaji, Paola Spoletini, Rajesh Gupta, Donatella Sciuto, and Yuvraj Agarwal. Buildingrules: A trigger-action based system to manage complex commercial buildings. In *Adjunct Proc. UbiComp*, 2015.

[60] Chandrakana Nandi and Michael D Ernst. Automatic trigger generation for rule-based smart homes. In *Proc. PLAS*, 2016.

[61] Robert P. Nix. Editing by example. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):600–621, 1985.

[62] openHAB. https://www.openhab.org/.

[63] Mitali Palekar, Earlence Fernandez, and Franziska Roesner. Analysis of the susceptibility of smart home programming interfaces to end user error. In *Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW)*, 2019.

[64] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive (1) designs. In *Proc. VMCAI*, 2006.

[65] Xin Qi, Qing Yang, David T. Nguyen, and Gang Zhou. 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*, 2013.

[66] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proc. ACL*, 2015.

[67] Parisa Rashidi and Diane J. Cook. Keeping the resident in the loop: Adapting the smart home to the user. *IEEE Trans. Systems, Man, and Cybernetics, Part A*, 39(5):949–959, 2009.

[68] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[69] Samsung. Capabilities reference. https://docs.smartthings.com/en/latest/capabilities-reference.html, Accessed February 2019.

[70] Maureen Schmitter-Edgecombe. Automated clinical assessment from smart home-based behavior data. *IEEE Journal of Biomedical and Health Informatics*, 2015.

[71] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[72] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *Proc. WWW*, 2017.

[73] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proc. CHI*, 2014.

[74] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes. In *Proc. CHI*, 2016.

[75] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. Charting the attack surface of trigger-action IoT platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[76] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the Internet of Things. In *Proc. NDSS*, 2018.

[77] Jong-bum Woo and Youn-kyung Lim. User experience in do-it-yourself-style smart homes. In *Proc. UbiComp*, 2015.

[78] Rayoung Yang and Mark W. Newman. 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*, 2013.

[79] Lana Yarosh and Pamela Zave. Locked or not?: Mental models of IoT feature interaction. In *Proc. CHI*, 2017.

[80] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. Autotap: Synthesizing and repairing trigger-action programs using ltl properties. In *Proceedings of the 41st International Conference on Software Engineering*, 2019.

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

[82] Sha Zhao, Julian Ramos, Jianrong Tao, Ziwen Jiang, Shijian Li, Zhaohui Wu, Gang Pan, and Anind K. Dey. 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*, 2016.

[83] Valerie Zhao, Lefan Zhang, Bo Wang, Michael L Littman, Shan Lu, and Blase Ur. Understanding trigger-action programs through novel visualizations of program differences. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2021.

[84] Valerie Zhao, Lefan Zhang, Bo Wang, Shan Lu, and Blase Ur. Visualizing differences to improve end-user understanding of trigger-action programs. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–10, 2020.