2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

AutoTap: Synthesizing and Repairing Trigger-Action Programs Using LTL Properties

Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, Blase Ur
Department of Computer Science, The University of Chicago
{lefanz, hewj, jessjmart, brackenburyn, shanlu, blase}@uchicago.edu

***Abstract*—End-user programming, particularly trigger-action programming (TAP), is a popular method of letting users express their intent for how smart devices and cloud services interact. Unfortunately, sometimes it can be challenging for users to correctly express their desires through TAP. This paper presents AutoTap, a system that lets novice users easily specify desired properties for devices and services. AutoTap translates these properties to linear temporal logic (LTL) and both automatically synthesizes property-satisfying TAP rules from scratch and repairs existing TAP rules. We designed AutoTap based on a user study about properties users wish to express. Through a second user study, we show that novice users made significantly fewer mistakes when expressing desired behaviors using AutoTap than using TAP rules. Our experiments show that AutoTap is a simple and effective option for expressive end-user programming.**

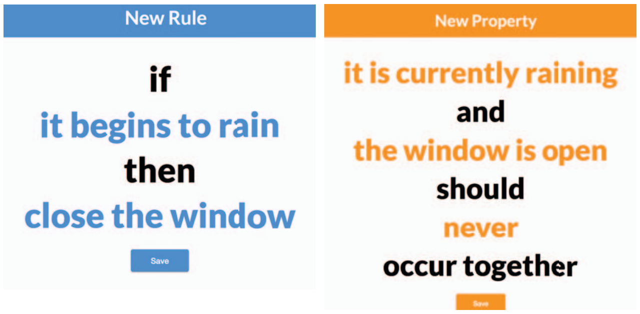
***Keywords*-End-user programming; trigger-action programming; program synthesis; program repair**

I. 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 [1], Mozilla's Things Gateway [2], Samsung SmartThings [3], Microsoft Flow [1], OpenHab [4], Home Assistant [5], Ripple [6], Zapier [1], and others. Some of these TAP services are widely used [7], [8].

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 [9], attempts to express more complex, yet commonly desired, behaviors often contain bugs [10]–[14]. These bugs encompass timing errors [10], issues with control flow [15], conflicting behaviors [12], and incorrect user expectations [14]. 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 [16] 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 1a). 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 requires a complex set of rules.



if
it begins to rain
then
close the window

it is currently raining
and
the window is open
should
never
occur together

(a) A (buggy) TAP *rule*. (b) A proposed TAP *property*.

Fig. 1: The TAP rule (a) cannot guarantee the property (b).

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 1b). 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 1b, 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:

1) A novel property-specification interface: The key goal of TAP is to empower novice users without programming knowl-

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00043

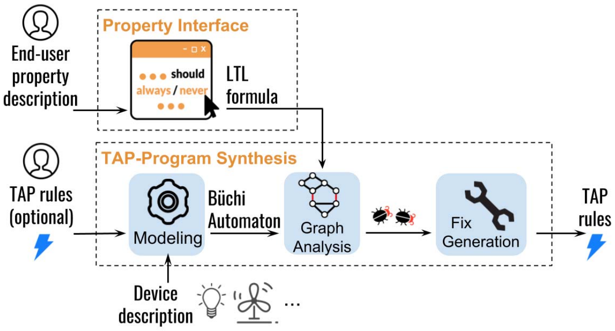


Fig. 2: An overview of AutoTap, which takes user-specified properties and (optionally) user-specified TAP rules to automatically generate a set of TAP rules that satisfy the properties.

edge 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 III). 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 [1]. 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 IV). While prior work has proposed interfaces for property specification [17], no prior efforts fully satisfy our requirements in the unique context of smart-device systems (Section VIII).

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 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 1b, 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. 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 V-A).

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. V-B).

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 V-C).

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 [8].

These techniques are also novel. We cannot use previously proposed synthesizers [18]–[20], which do not satisfy the requirements discussed above in the unique context of smart-device systems (Section VIII). A small but quickly growing literature has begun to apply formal methods to TAP [21]–[24]. Our techniques move beyond this work in both the target and the solution. Some of this work only aims to detect property violations [24], while others only repair *existing* rules by editing or adding conditions [21], [22] or triggers [23]. 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], [25].

Our evaluation of AutoTap includes several parts (Section VI). 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.

II. Background

A. Trigger-Action Programming (TAP)

In recent years, TAP has received a great deal of academic attention in multiple areas: usability [9]–[11], [13]; novel interfaces [26]–[28]; measurement [7], [8]; deployment [6], [29]; correctness [12], [21]–[24]; and security [15], [16], [30]. Furthermore, TAP has been deployed by Microsoft [1], Mozilla [2], IFTTT [1], Samsung [3], and others.

Systems generally follow one of two TAP rule structures [14]. The simpler variant connects a single trigger to a single action: "IF *event* THEN *action*." Each such statement is a TAP *rule*, and a collection of rules forms a TAP *program*. *Events* include state changes for devices, services, and sensors (e.g., "it begins to rain"). *Actions* are actuations of devices (e.g., "open the window") or services (e.g., "send an SMS").

The more expressive variant differentiates **events** (actions or state changes that occur in a moment, such as "it begins to rain") and **states** (conditions that remains true/false over time, such as "it is raining"). In this variant, triggers are a single event optionally conditioned on one or more states as follows: "IF *event* occurs WHILE devices are in a given *state*, THEN fire *action*," shortened as "IF *event* WHILE *state(s)* THEN *action*." In this paper, we use this more expressive EVENT-STATE-ACTION variant (also called EVENT-CONDITION-ACTION), which balances usability and expressiveness. This variant is used in Samsung SmartRules [3], Stringify [31], Home Assistant [5], and academic studies [9], [10], [14], [32].

B. Transition Systems and Linear Temporal Logic (LTL)

AutoTap formally models smart devices and TAP programs as transition systems [33]. Every transition system consists of a set of states S; a set of events E (typically called actions in TAP) that change the system from one state to another, $s_1 \xrightarrow{e \in E} s_2$; and a set of atomic propositions AP that reflect detailed properties of a state, with $L(s)$ denoting the set of atomic propositions associated with state s. A valid execution is an infinite sequence of states $s_0 s_1 ...$, $s_k \in S$ and every transition from one state to the next is valid, $s_i \xrightarrow{e_i \in E} s_{i+1}$.

LTL formulas can represent a wide variety of execution properties and are widely used in formal verification [34]. An LTL formula ϕ is constructed from atomic propositions with some operators: $\phi ::= \text{true}|ap|\neg\phi|\phi_1 \wedge \phi_2|\mathbf{X}\phi|\phi_1\mathbf{U}\phi_2$. Informally, $\neg, \wedge, \mathbf{X}$ and $\mathbf{U}$ represent *not*, *and*, *neXt*, and ***U**ntil*, respectively. In addition to these basic operators, $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{W}$ are common derived operators. Given an execution $E = s_0 s_1 ...$ of a transition system, E satisfies atomic proposition ap if and only if the initial state of E is associated with ap (i.e., $ap \in L(s_0)$). A transition system TS satisfies property ϕ ($TS \models \phi$) if all possible executions in TS satisfy ϕ.

III. 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 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.

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

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

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

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

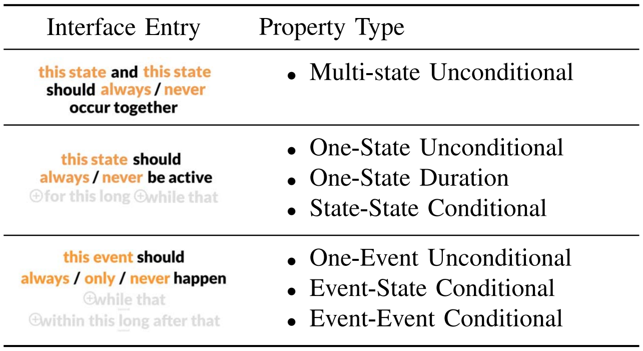
e) ***State-State Conditional (1.6%, 7.5%):*** "The stove should always be off if no one is home."

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

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

TABLE I: 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. V-A).

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 $[event_2]$	$\mathbf{G}(@event2 \rightarrow (time\#event2\mathbf{W}@event1))$ $\neg\mathbf{F}(time\#event_2 \wedge \mathbf{X}@event_1)$



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

Fig. 3: Templates in AutoTap's property-specification UI.

IV. 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 I 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 I. The example in Figure 1a 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 3, 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 1.

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 thus can be handled by AutoTap.

V. 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 4: (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.

The second step can be done through model checking [21], 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.

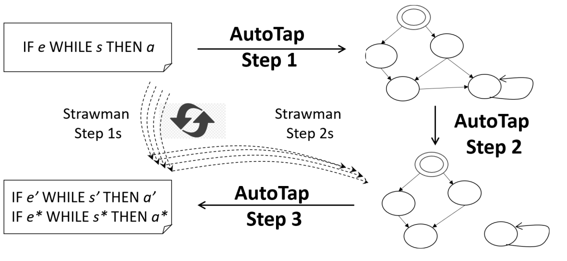

IF e WHILE s THEN a
AutoTap Step 1
Strawman Step 1s
Strawman Step 2s
AutoTap Step 2
IF e' WHILE s' THEN a'
IF e^* WHILE s^* THEN a^*
AutoTap Step 3

Fig. 4: AutoTap approach vs. straw-man approach

AutoTap approach: AutoTap takes a unique approach to solving this problem in a general and systematic way. As illustrated in Figure 4, 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 V-A 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 V-B 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 IV 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 V-C, it can be done in a systematic way for TAP.

A. Step 1: Model Construction

AutoTap's inputs are: (1) safety properties ϕ in LTL, obtained through the user interface presented in Section IV; (2) TAP rules, if any; (3) specifications for every smart device in the form of a transition system, as defined in Section II-B. 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 [35].

AutoTap's baseline model construction follows traditional model-checking techniques [36]. First, a transition system is built for a set of devices together with their TAP rules, if any (e.g., Figure 5). 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 6b). Next, AutoTap applies Spot [37] 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 6a). 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., Figure 7).

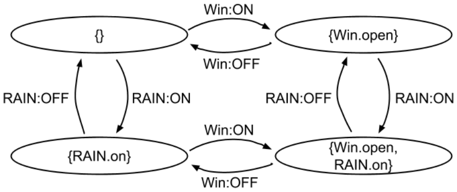


Fig. 5: Transition system for **RAIN** and a **Win**dow. Statements in parentheses are Atomic Propositions held in each state.

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

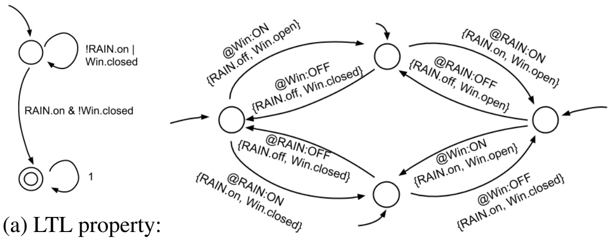
Device selection: To avoid unnecessary complexity, AutoTap selects devices D related to the given property ϕ to model. To do so, AutoTap first initializes D with all the devices that appear in ϕ. AutoTap then iteratively expands D with devices that can affect any device already in D until reaching a 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 $\mathrm{timer}(t\#e)$ or $\mathrm{timer}(t{*}ap)$ 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 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-

[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 [35] contains such information: capabilities with "commands" are controllable, while others can only be sensed.



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

(b) Device system: Window + RAIN

Fig. 6: Büchi Automata of our running example.

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 [38] and more complicated timed automata [39] will only add unnecessary complexity to AutoTap property checking and rule synthesis.

B. 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 [36], 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 6a 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 node in Figure 6a.

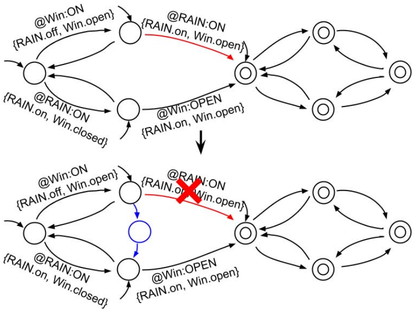


Fig. 7: Combined Büchi Automaton of the running example. (The top is the original. The bottom is after adding a rule.)

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 7.

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, after 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 V-B, 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.

C. 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 :

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

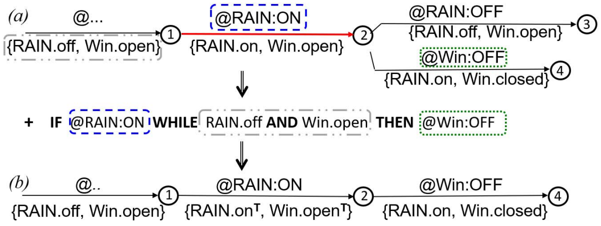


Fig. 8: Device automaton (a) changed to (b) by adding a rule.

$n1 \xrightarrow{ap} n2$ 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 8a is part of the automaton $\mathbb{A}_s$ in Figure 6b 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 6a. 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 7.

Figure 8b 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 8. By changing the nature of ①→②, its AP-set no longer matches with that of the bridge edge in Figure 6a. Consequently, the corresponding bridge edge in $\mathbb{A}$ (i.e., the red edge in Figure 7) will disappear.

1) 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 9a. 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 8, 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.

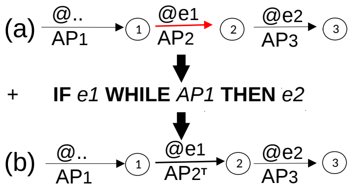


Fig. 9: Generalization of adding TAP rules.

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 8 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 9. 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 V-A), 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.

VI. Evaluation

A. *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 [8] and prior usability studies have shown that even novice users can create TAP rules successfully [9], [13], [28], [40].

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 1a).
- **Properties**: Participants created properties using AutoTap's interface (see Figure 1b)[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 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.

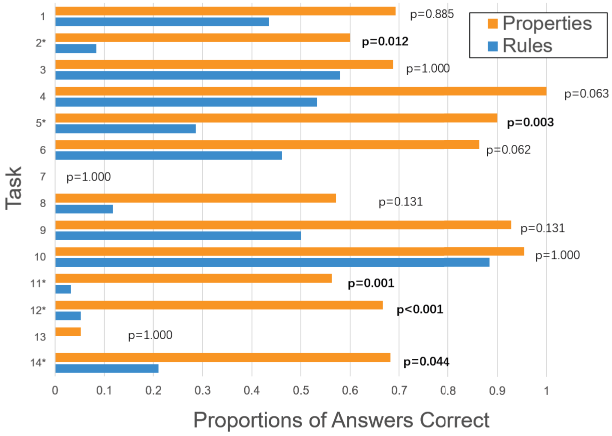


Fig. 10: Correctness of properties and rules by task. P-values are from Holm-corrected χ^2 tests comparing the proportion of statements correct when written using rules versus properties.

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 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

[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 I and Figure 3. 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.

TAP rule interface. This difference was statistically significant for five of these tasks (the bolded p-values in Figure 10). 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 10), 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 $U = 590.5$, $p = .052$).

B. 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 I. 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.

TABLE II: 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}

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.

C. 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 II. 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 II, 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.

D. 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."

VII. Threats to AutoTap's Validity

AutoTap is not guaranteed to generate patches for every LTL safety property. Patches are generated by the current prototype of AutoTap when (1) bridge edges are found, and (2) the bridge can be cut with a single TAP rule. The 1 out of 158 cases where AutoTap fails to synthesize a TAP program in Section VI-B violates the second assumption. The 2 out of 25 cases where AutoTap fails to fix a TAP program in Section VI-C violates the first assumption. Both limitations can be fixed by future extensions to AutoTap. Furthermore, the first assumption does not hold if every state is accepting, meaning that no matter what actions we take in the system, we cannot prevent it from triggering a violation. The second assumption does not hold when there are no controllable actions to escape from a property violation. That is, only events out of our control (e.g, changing the weather) help. These scenarios occur when the system lacks critical functionality or the property itself is conflicting, which is out of scope for AutoTap.

We focus on TAP instead of other smart-device languages mainly because TAP is widely used [8] and easy for end-users to understand [9]. AutoTap is not limited to TAP. Cutting bridge edges that cause property violations can be accomplished in other automation languages, too. In fact, we feel that some bridges might be better fixed by "disabling rules" that can conditionally disable actions.

AutoTap currently does not consider issues like actions failing to complete or not taking effect immediately [14]. Handling these issues requires device manufacturers to provide a more accurate model of the system. Furthermore, users can still make mistakes in writing properties. Their properties might not reflect their real intent. Properties could even conflict with each other, which AutoTap does not currently resolve.

VIII. Related Work

TAP program bug-detection and fixing: AutoTap is inspired by previous work [21], [22] 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 [22] only accepts "[$states$] shall not happen", missing many common desires.

TrigGen [23] detects a specific type of bug in OpenHAB TAP programs [4] – 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 [28] or synthesizing TAP rules from natural language [26], [27]. Our synthesis and repair techniques are complementary to those techniques.

Program synthesis using formal methods: Synthesizing a program from a formal specification, or *LTL synthesis*, has been an open problem [34]. Most work in this area synthesizes reactive systems based on formal specifications [18]–[20], [34]. 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 [25]. 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.

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 [41]. 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 [17]. 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.

IX. 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.

X. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCF-1837120, OAC-1835890, CNS-1764039, CNS-1563956, CNS-1514256, and IIS-1546543, as well as gifts from the CERES Center. We thank Abhimanyu Deora for help with the user interface and Roshni Padhi for help with the user studies.

References

[1] T. Klosowski, "Automation showdown: IFTTT vs Zapier vs Microsoft Flow," LifeHacker, June 26, 2016.

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

[3] W. Mossberg, "SmartThings automates your house via sensors, app," Recode.net, 2014.

[4] openHAB, https://www.openhab.org/.

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

[6] R. Chard, K. Chard, J. Alt, D. Y. Parkinson, S. Tuecke, and I. Foster, "Ripple: Home automation for research data management," in *Proc. ICDCSW*, 2017.

[7] B. Ur, M. Pak Yong Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman, "Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes," in *Proc. CHI*, 2016.

[8] X. Mi, F. Qian, Y. Zhang, and X. Wang, "An empirical characterization of IFTTT: Ecosystem, usage, and performance," in *Proc. IMC*, 2017.

[9] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, "Practical trigger-action programming in the smart home," in *Proc. CHI*, 2014.

[10] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *Proc. UbiComp*, 2015.

[11] L. Yarosh and P. Zave, "Locked or not?: Mental models of IoT feature interaction," in *Proc. CHI*, 2017.

[12] A. A. Nacci, B. Balaji, P. Spoletini, R. Gupta, D. Sciuto, and Y. Agarwal, "Buildingrules: A trigger-action based system to manage complex commercial buildings," in *Adjunct Proc. UbiComp*, 2015.

[13] J. Brich, M. Walch, M. Rietzler, M. Weber, and F. Schaub, "Exploring end user programming needs in home automation," *ACM TOCHI*, vol. 24, no. 2, p. 11, 2017.

[14] W. Brackenbury, A. Deora, J. Ritchey, J. Vallee, W. He, G. Wang, M. L. Littman, and B. Ur, "How users interpret bugs in trigger-action programming," in *Proc. CHI*, 2019.

[15] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the Internet of Things," in *Proc. NDSS*, 2018.

[16] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia, "Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes," in *Proc. WWW*, 2017.

[17] M. Lumpe, I. Meedeniya, and L. Grunske, "PSPWizard: machine-assisted definition of temporal logical properties with specification patterns," in *Proc. ESEC/FSE*, 2011.

[18] J. R. Büchi and L. H. Landweber, "Solving sequential conditions by finite-state strategies," *Transactions of the American Mathematical Society*, vol. 138, pp. 295–311, 1969.

[19] N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of reactive (1) designs," in *Proc. VMCAI*, 2006.

[20] E. Letier and W. Heaven, "Requirements modelling by synthesis of deontic input-output automata," in *Proc. ICSE*, 2013.

[21] C.-J. M. Liang, L. Bu, Z. Li, J. Zhang, S. Han, B. F. Karlsson, D. Zhang, and F. Zhao, "Systematically debugging IoT control system correctness for building automation," in *Proc. BuildSys*, 2016.

[22] L. Bu, W. Xiong, C.-J. M. Liang, S. Han, D. Zhang, S. Lin, and X. Li, "Systematically ensuring the confidence of real-time home automation IoT systems," *ACM TCPS*, vol. 2, no. 3, p. 22, 2018.

[23] C. Nandi and M. D. Ernst, "Automatic trigger generation for rule-based smart homes," in *Proc. PLAS*, 2016.

[24] Z. B. Celik, P. McDaniel, and G. Tan, "SOTERIA: Automated IoT safety and security analysis," in *Proc. USENIX ATC*, 2018.

[25] R. Degiovanni, D. Alrajeh, N. Aguirre, and S. Uchitel, "Automated goal operationalisation based on interpolation and SAT solving," in *Proc. ICSE*, 2014.

[26] X. Chen, C. Liu, R. Shin, D. Song, and M. Chen, "Latent attention for if-then program synthesis," in *Proc. NIPS*, 2016.

[27] C. Quirk, R. Mooney, and M. Galley, "Language to code: Learning semantic parsers for if-this-then-that recipes," in *Proc. ACL*, 2015.

[28] T.-H. K. Huang, A. Azaria, and J. P. Bigham, "Instructablecrowd: Creating if-then rules via conversations with the crowd," in *Proc. CHI Extended Abstracts*, 2016.

[29] J.-b. Woo and Y.-k. Lim, "User experience in do-it-yourself-style smart homes," in *Proc. UbiComp*, 2015.

[30] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action IoT platforms," in *Proc. NDSS*, 2018.

[31] E. Oswald, "IFTTT competitor Stringify gets a major update," TechHive, June 22, 2016.

[32] A. Rahmati, E. Fernandes, J. Jung, and A. Prakash, "IFTTT vs. Zapier: A comparative study of trigger-action programming frameworks," *arXiv:1709.02788*, 2017.

[33] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[34] R. Bodik and B. Jobstmann, "Algorithmic program synthesis: Introduction," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 397–411, Oct 2013.

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

[36] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Proc. PSTV*, 1995.

[37] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0a framework for ltl and ω-automata manipulation," in *Proc. ATVA*, 2016.

[38] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

[39] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[40] G. Ghiani, M. Manca, F. Paternò, and C. Santoro, "Personalization of context-dependent applications through trigger-action rules," *ACM TOCHI*, vol. 24, no. 2, p. 14, 2017.

[41] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: An environment for goal-driven requirements engineering," in *Proc. ICSE*, 1997.