Understanding Trigger-Action Programs Through Novel Visualizations of Program Differences

Valerie Zhao
University of Chicago
United States
vzhao@uchicago.edu

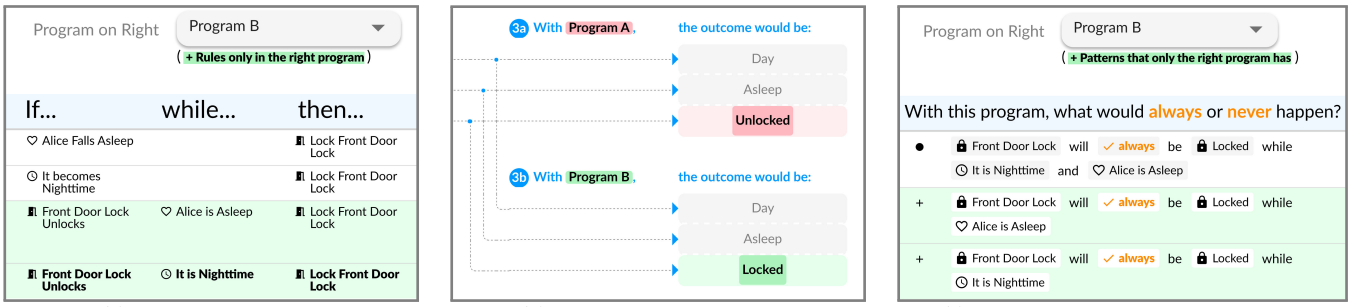
Lefan Zhang
University of Chicago
United States
lefanz@uchicago.edu

Bo Wang
National University of Singapore
Singapore
bo_wang@u.nus.edu

Michael L. Littman
Brown University
United States
mlittman@cs.brown.edu

Shan Lu
University of Chicago
United States
shanlu@uchicago.edu

Blase Ur
University of Chicago
United States
blase@uchicago.edu



(a) Differences in text. **(b) Differences in outcomes.** **(c) Differences in abstract properties.**

Figure 1: These snippets of TAP diff interfaces we designed illustrate the traditional approach of showing differences in program text (left) and our more novel approaches of showing differences in program outcomes (center) and properties (right).

ABSTRACT

Trigger-action programming (if-this-then-that rules) empowers non-technical users to automate services and smart devices. As a user's set of trigger-action programs evolves, the user must reason about behavior differences between similar programs, such as between an original program and several modification candidates, to select programs that meet their goals. To facilitate this process, we co-designed user interfaces and underlying algorithms to highlight differences between trigger-action programs. Our novel approaches leverage formal methods to efficiently identify and visualize differences in program *outcomes* or abstract *properties*. We also implemented a traditional interface that shows only syntax differences in the rules themselves. In a between-subjects online experiment with 107 participants, 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.

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05.
https://doi.org/10.1145/3411764.3445567

CCS CONCEPTS

• **Human-centered computing** → **Interaction design**; **Empirical studies in ubiquitous and mobile computing**; **User interface programming**.

KEYWORDS

trigger-action programs, end-user programming, diffs, debugging

ACM Reference Format:
Valerie Zhao, Lefan Zhang, Bo Wang, Michael L. Littman, Shan Lu, and Blase Ur. 2021. Understanding Trigger-Action Programs Through Novel Visualizations of Program Differences. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3411764.3445567

1 INTRODUCTION

Trigger-action programming (**TAP**) is a paradigm for end-user development and composition in which users create **rules** of the form "IF [trigger] WHILE [conditions] THEN [action]" using a graphical interface. For example, the rule "IF Alice falls asleep WHILE it is nighttime AND the front door is unlocked THEN lock the front door" instructs the home to lock the front door when Alice falls asleep at night. We term a set of TAP rules a TAP **program**. TAP has shown promise in empowering non-technical users to connect and automate Internet-of-Things devices and online services [46]. It underpins end-user automation in domains including social media [44], business [49], scientific research [7, 8], and smart

homes [30, 43, 47]. Services that support TAP include IFTTT [19], Microsoft Flow [32], Zapier [49], and Mozilla WebThings [13, 31].

TAP is intuitive and easy to use [14, 46], but it is also vulnerable to reasoning errors [4, 17, 48]. Complex interactions between rules and device states can hinder users from knowing precisely when the TAP system would initiate an action [4, 48]. Increasingly large deployments will exacerbate TAP complexity by requiring many rules to satisfy potentially conflicting requirements. This complexity makes it challenging for users to write rules that match their intent or to debug programs, leading to problems ranging from discomfort to wasted resources to security risks.

During the development and maintenance of TAP programs, comparing similar TAP programs is a common and crucial task. We use the term **variants** to refer to highly similar programs being compared. When a user is iteratively tweaking or debugging their program either to add new behaviors or to fix bugs, the original program and the tweaked programs the user creates are variants. Because TAP programs can be long (contain many rules) or be complex in the degree to which different rules act on the same device, users might modify a program and struggle to determine whether they have achieved their goal. Similarly, taking advantage of ecosystems of shared TAP programs [19], users might merge programs created by others into their own existing programs, wondering how the variant reflecting the combined programs compares to the original. Furthermore, researchers have recently created tools for automatically synthesizing TAP programs [25, 50, 51]. These methods often output multiple candidate variants from a single input, leaving the user wondering which variant to choose.

We further illustrate instances where users may wish to compare TAP programs with the following vignette. Suppose Alice uses Program A (Figure 2) for her home. She expects the door to be locked whenever she is asleep or whenever it is nighttime. However, she sometimes notices that the door is unlocked at night, making her concerned about the safety of her home. Alice decides to fix this problem by modifying Program A into Program B (Figure 2). With our tool, she can now compare the two programs to determine whether Program B resolves the issue Program A had. Her visiting mother distrusts Program B and instead proposes Program C (Figure 2). Alice can again compare Program B with Program C to see that both programs behave identically.

To help end-users correctly reason about TAP, we designed and evaluated a set of novel user interfaces, powered by formal analysis of TAP programs, that compare TAP programs not just syntactically, but *semantically.* Although our methods can be applied to any arbitrary set of programs, we focus on highly similar variants because the number of differences will be relatively small and therefore tractable to surface to users. While previous work sought to improve TAP understanding by visualizing previous and potential smart home behaviors [9, 28] or explaining certain types of programming errors [10, 11, 26], our interfaces helps users more broadly understand the effects of changes while modifying programs or while selecting among variants to meet their goals.

Traditional **diff**[1] interfaces like those on GitHub or Google Docs compare program text [2, 24, 36, 42]. Our semantic-diff interfaces differentiate programs based on *situational outcomes, general properties,* or *user selection of desired outcomes.* Figure 1 highlights these different ways of reasoning about the differences between example Programs A and B (Figure 2). Furthermore, whereas a traditional text-diff interface would highlight syntax differences between Programs B and C (Figure 2), our outcome-diff interface would show that the programs actually behave identically. Because these novel semantic-diff interfaces are driven by our formal analysis approach that leverages the relatively small state space of TAP programs, they are specific to TAP, although the concepts can perhaps be adapted to other constrained applications.

To evaluate the effectiveness of these interfaces, we conducted a 107-participant online experiment. We randomly assigned each participant to an interface—either one of our diff interfaces or a control interface showing just the programs themselves. While the control interface and traditional text-diff interface enabled participants to identify differences in short and straightforward programs, our novel semantic-diff interfaces significantly outperformed those interfaces when participants aimed to identify differences between long and complex programs. Notably, our novel interfaces focused on outcomes helped participants accurately complete a wide variety of tasks, and our interface focused on high-level properties helped participants overlook low-level details when appropriate. While (compared to our controls) our novel interfaces helped a significantly larger fraction of participants correctly identify differences between long, complex programs, the time it took to complete tasks did not differ significantly across interfaces.

The paper proceeds as follows. Section 2 defines our terminology and assumptions, while Section 3 presents related work. We motivate and describe the interaction design of our user interfaces in Section 4, and we then describe our formal model of TAP systems and novel algorithms underpinning and enabling these interfaces in Section 5. We present the methodology of our user study in Section 6 and the results in Section 7. In Section 8, we discuss implications and deployment considerations for TAP systems.

2 TERMINOLOGY AND DEFINITIONS

To facilitate our presentation of this work, we define the following terms and note the following assumptions. As mentioned in Section 1, we term a set of TAP rules a **program**. We term the related programs being compared with our interface **variants**.

A **factor** is any element relevant to the smart home system, including smart devices, the users, and the environment (e.g., weather, time of day). An **attribute** is an aspect of a factor. Example attributes of a user are whether they are asleep and whether they are at home. We will refer to the pair of a factor and an attribute as a **variable** (e.g., "whether the front door is locked" and "whether Alice is asleep"). A **state** is a statement that is true about the variable over some period of time (e.g., "Alice is asleep"), while an **event** is an instantaneous change in the variable's state (e.g., "Alice falls asleep"). A **smart home system state** is the set of all variable states in an environment at that point in time.

In its simplest form, a TAP rule takes the form "IF [trigger] THEN [action]." The **action** is an event that can be automated, such as locking a smart door lock. The trigger activates the action. Out of several trigger types, we use event-state triggers in this work because prior work [4, 17, 38] found this type to be most intuitive

[1] Many tools for comparing differences use the name "diff," by analogy to the Unix diff utility [18]. We call any comparison interface a "diff" interface.

Program A:

(1) **IF** Alice falls asleep **THEN** lock the front door.
(2) **IF** it becomes nighttime **THEN** lock the front door.
(3) **IF** the front door unlocks **WHILE** Alice is asleep **AND** it is nighttime **THEN** lock the front door.

Program B:

(1) **IF** Alice falls asleep **THEN** lock the front door.
(2) **IF** it becomes nighttime **THEN** lock the front door.
(3) **IF** the front door unlocks **WHILE** Alice is asleep **THEN** lock the front door.
(4) **IF** the front door unlocks **WHILE** it is nighttime **THEN** lock the front door.

Program C:

(1) **IF** Alice falls asleep **WHILE** it is daytime **THEN** lock the front door.
(2) **IF** it becomes nighttime **THEN** lock the front door.
(3) **IF** the front door unlocks **WHILE** Alice is asleep **THEN** lock the front door.
(4) **IF** the front door unlocks **WHILE** Alice is awake **AND** it is nighttime **THEN** lock the front door.

Figure 2: A running example. Programs A, B, and C all have different rules, but only Program A behaves differently from the others. Program A allows the front door to unlock and stay unlocked when Alice is awake at night or when she is asleep during the day. Meanwhile, Programs B and C both ensure the front door remains locked while Alice is asleep and/or it is nighttime.

for TAP. Event-state triggers consist of an event and zero or more states that must all be true to trigger the action. Note that we refer to just the event part of an event-state trigger as the **trigger** and the states (if any) as the set of **conditions**, such that a rule may have the form "IF [trigger] WHILE [conditions] THEN [action]." We refer to the trigger, the conditions, and the action as rule **subparts**.

For rules whose action reverses the trigger (e.g., "IF the front door unlocks THEN lock the front door," as in Figure 2), we assume the rule *prevents* the trigger from occurring, rather than allowing the trigger to proceed and then immediately reversing it. We made this decision to minimize confusion about whether an action occurs.

3 RELATED WORK

Reasoning Errors in TAP. Although users find TAP easy to use, they are susceptible to errors in reasoning. Yarosh and Zave [48] found users struggle to reason about smart home feature interactions and determine rule outcomes, partially inspiring our outcome-based interface. Huang and Cakmak [17] identified errors in users' mental models of different types of TAP triggers and behaviors, including in reasoning about the temporality of different behaviors. Brackenbury et al. [4] identified additional errors, such as misconceptions about actions automatically reverting. In visits to households with smart home systems, Brush et al. [5] noted some users find TAP difficult to debug and had stopped using such rules or resigned themselves to living with the bugs. More recently, Palekar et al. [38] concluded many current TAP interfaces are not foolproof against bugs. Some of these bugs result from misalignment between a user's intent and the program they write; these gaps *cannot* be automatically inferred. Users might omit triggers, conditions, or actions they had in mind, motivating our approach to co-design diff interfaces and underlying algorithms to highlight TAP outcomes.

End-User Programming Support. Prior work has developed tools to support various end-user programming paradigms [21, 33]. Well-known examples include Whyline for the Alice language [22], which anticipates users asking "why a bug has occurred" by offering explanations upon the appearance of bugs. Even though end-user program variants are the natural products of program evolution and modification, comparatively less work has focused on them. Kuttal et al. [23] investigated end-user programming variant management practices. They focused on managing multiple variants, whereas we identify and present differences between variants.

Some recent interfaces help users reason about TAP and smart home automation, though we are the first to highlight differences in outcomes or properties across TAP variants. EUDebug [10] and My IoT Puzzle [11] identify TAP bugs—specifically inconsistent rules, redundant rules, and rules that cause loops. ITAD [26] aids end-user TAP debugging by simulating program behavior and identifying rule conflicts. We instead leverage algorithms and visualizations to identify and explain *variations* and *differences* across programs. Doing so is critical to minimizing gaps between a user's intent and their TAP programs. Our interfaces support debugging, as well as modifying or selecting between candidate programs. EUDebug [10], My IoT Puzzle [11], and our work all incorporate flowcharts in some cases. The prior work focuses on illustrating the events at each step, whereas our flowcharts focus on the effects of the rules in diverging outcomes among program variants.

Other work helps users better understand aspects of their smart home's rules. Casalendar [28] visualizes past events caused by the user's TAP programs in a calendar interface, contextualizing programs' effects relative to the user's schedule. Castelli et al. [6] developed a dashboard system that helps users interpret logged sensor readings to better understand energy consumption. FORTNIoT [9] uses simulations to predict when the rules of a TAP program might execute in the future under certain assumptions, but does not support comparing programs. We instead help users compare TAP variants in terms of their diverging future behaviors. Although FORTNIoT and our *Outcome-Diff: Flowcharts* interface both visualize system behavior under a concrete situation, we analyze state machines representing multiple TAP variants and comprehensively identify all situations leading to different behaviors. Common software engineering activities, namely requirements definition, specification, code reuse, verification, and debugging, also occur (sometimes implicitly) in end-user programming [21]. Our interfaces assist TAP users in the latter three activities.

Diffs in Other Domains. Tools in other domains, often modeled after the Unix diff utility [18], also present text differences for program versioning or collaborative writing [34]. Interface examples include the "split diff" and "unified diff" views on GitHub [36], the version history interface on Google Docs [42], and the online tool Mergely [39]. These tools typically either present variants side-by-side with differences aligned (an example is shown in our supplementary materials [53]) or superimposed. Surprisingly little work has evaluated these approaches' usability. Other work analyzes software similarity for non-user-facing applications, such as

detecting code reuse [16, 20], plagiarism [35], or clustering similar student programs for large classes [15]. Some work uses neural networks to measure functional similarity [52].

Work on supporting student programmers has extracted and visualized semantic differences as well, although it has not been applied to TAP. TraceDiff [45] dynamically extracts trace differences between an incorrect student program and a similar, correct solution. Although its theme of visualizing behavior difference is similar to our *Outcome-Diff: Flowcharts*, TraceDiff relies on a predetermined suite of test cases as program inputs to check for behavioral differences. We instead automatically identify inputs (system state and events) from the transition systems that would result in behavioral differences. We also visualize outcome differences as isolated steps, one per discrete situation, as opposed to a series of steps over a period of time as a trace. AutomataTutor [12] provides high-level hints for, among other aspects, syntactic mistakes in deterministic finite automaton (DFA) construction by identifying the DFA edit distance [1] from the student's incorrect solution to a correct solution. It provides counterexamples to demonstrate errors in the incorrect solution. Our *Property-Diff* algorithm also extracts high-level information about transition systems, but we focus on unreachable nodes and self-transitions to identify statements about what can always or never happen in the system. We do not leverage all differences between the graphs, nor make statements about what the system can accept.

4 DIFF-BASED USER INTERFACES

In this section, we present our interfaces and describe their goals and interaction design. Section 5 complements this section by describing the algorithms underpinning these interfaces. We first discuss our control conditions, *Rules* and *Text-Diff*. The former shows just the programs themselves, while the latter highlights syntax differences. We then describe our semantic-diff interfaces that emphasize TAP differences beyond syntax. We present two interfaces that visualize differences in behavior outcomes between variants (*Outcome-Diff: Flowcharts* and *Outcome-Diff: Questions*) and one that compares abstract properties guaranteed by variants (*Property-Diff*). For each interface, we show how it compares Programs A and B from our running example (Figure 2).

4.1 Control Conditions

The *Rules* interface displays each program individually (Figure 3). Existing systems show trigger-action programs to users individually, so this approach serves as a control condition against diff interfaces. We expected this traditional interface to be sufficient for short, simple variants. A dropdown menu at the top allows users to toggle which program they see. Each row lists a rule with its subparts (trigger, conditions, and action) separated into columns.

The *Text-Diff* interface displays programs side-by-side. It highlights rules that differ, as in traditional diff interfaces (Figure 4). We expected *Text-Diff* to help contrast TAP variants that have a small number of key differences. For example, if the differences are simple and independent from the other rules (e.g., if the differences concern the TV and lights, while all other rules relate to the front door lock and whether Alice is asleep), we expected

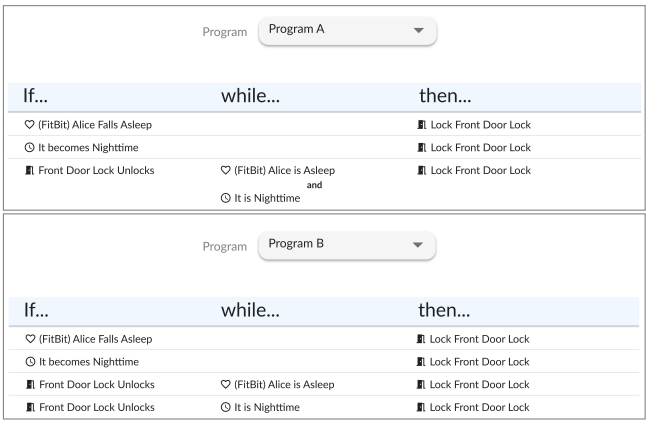


Figure 3: *Rules* of Program A (left) and Program B (right) from Figure 2. Dropdown menus toggle the program shown.

Text-Diff to be sufficient. Unfortunately, for realistic and practical TAP applications, the text of programs can differ substantially.

We chose *Text-Diff* as a control condition to represent existing diff tools. We piloted several implementations representative of popular tools like GitHub's "split diff" and "unified diff" [36], as well as Google Docs version history changes [42]. Pilot participants preferred the "split diff" design, so we adopted it as our final design.

Like GitHub's "split diff," our *Text-Diff* describes how one can modify the first variant to become the second. Rules unique to the first variant are "deleted" (shown with a minus sign and red background). Rules unique to the second variant are "added" (with a plus sign and green background). The plus and minus signs mitigate the potential inaccessibility of a red/green color scheme to colorblind users. We also bold the text differences. A rule in the first variant is instead "modified" into a rule in the second if they are similar. We align such rules on the same row and show precise differences with a darker red/green background. In Figure 4, the third rule on the left is "modified" to become the third rule on the right by removing "AND it is nighttime." Another rule, "IF front door unlocks WHILE it is nighttime THEN lock the front door," is "added" as the fourth rule on the right. When there are more than two variants, dropdown menus let users select which two to compare.

We redesigned parts of GitHub's "split diff" approach to suit TAP. For a rule to be considered "modified," we defined that they must differ in exactly one subpart. To reduce confusion from misaligned rules, we also sorted the rules of the second variant relative to the first. Unlike traditional software or text documents, trigger-action programs are sets of rules in which the order is flexible. If two variants have identical rules in different orders, a traditional code-diff interface would thus highlight them as different.

4.2 Outcome Differences

A shortcoming of traditional diff interfaces is that they do not help users directly discern when a system would take specific actions, nor how the variants cause these actions to differ. By seeing or comparing just the text of Programs A and B from Figure 2, a user might fail to recognize that when someone attempts to unlock the front door while Alice is asleep during the day, the smart home

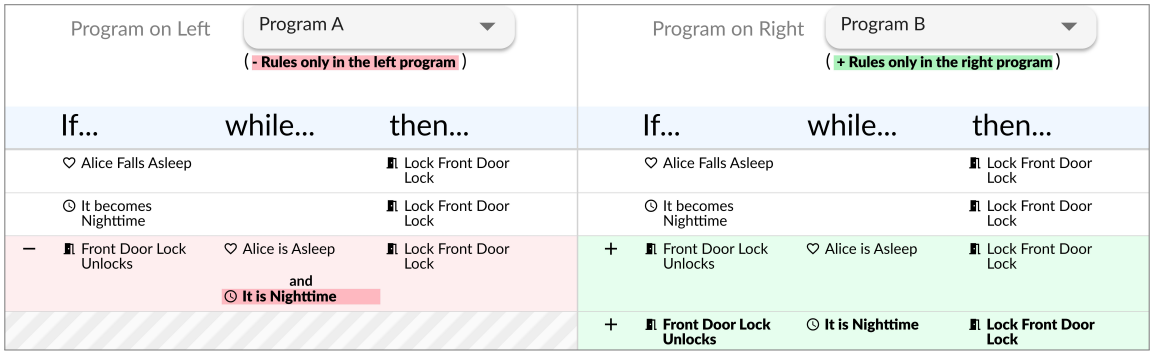


Figure 4: *Text-Diff* comparing Programs A and B from Figure 2. Similar to GitHub's "split diff" [36], rules unique to a variant have a color background and are preceded by a symbol (red and "-" on the left, green and "+" on the right). When a pair of rules from the two variants is similar, they are aligned on the same row with differences highlighted in darker red or green.

will prevent them from doing so with Program B, but not A. It is critical to identify the situations in which the variants differ in behavior, but examining only the rules themselves can obscure this information. Here, the *situation* is the context of Alice being asleep while it is daytime and someone is trying to unlock the front door. The *outcome* of Program A is that the front door is unlocked, while with Program B it is locked. Variants produce different outcomes when they take different actions under identical situations.

To help users reason about program differences when they have in mind desired outcomes under specific situations, we present two interfaces that visualize outcome differences. *Outcome-Diff: Flowcharts* uses flowcharts to display all situations in which outcomes differ. *Outcome-Diff: Questions* asks the user to select their desired outcome(s) for these situations through checkboxes, and then summarizes how many, and which, selected outcomes occur under each program variant. To avoid overloading the user, neither interface shows situations with identical outcomes.

***Outcome-Diff: Flowcharts* shows all situations in which two variants produce different outcomes** (Figure 5). Via dropdown menus, a user chooses two variants to compare and sees a series of flowcharts highlighting differences in outcomes. The interface also states the number of situations in which outcomes differ between those variants. Each flowchart shows a situation. For Programs A and B, *Outcome-Diff: Flowcharts* shows two situations in which the two programs produce different outcomes. The first situation (Figure 5) starts with Alice asleep during the day while the front door is locked. Then, a hypothetical event—the front door unlocking—occurs. Program A results in the front door being *unlocked*, while Program B results in it being *locked*. Variable state differences use the same color scheme as text differences in *Text-Diff*, and we took the same steps to enhance accessibility.

***Outcome-Diff: Questions* instead asks the user to indicate what outcomes, if any, are desirable in particular situations**, as shown in Figure 6. It shows the same situations as *Outcome-Diff: Flowcharts*, albeit in this revised question format. Whereas *Outcome-Diff: Flowcharts*, *Text-Diff*, and *Property-Diff* all compare exactly two variants at a time, pairwise comparisons are tedious and overwhelming if there are many variants to compare. The approach of *Outcome-Diff: Questions* enables the user to quickly identify variants with the desired behaviors among a large set of variants. For each specific situation, the interface lists all of the different outcomes caused by at least one of the variants under that situation. The user selects their desired choice(s), or specifies that they have no preference. Figure 6a shows an example of the same situation in Figure 5 for Programs A and B. Below the situation are the outcome choices. Differences between outcomes have orange backgrounds in selected outcomes and blue backgrounds in unselected outcomes.

For each program variant, *Outcome-Diff: Questions* tracks the number of situations in which the user-selected outcome matches what would occur in that variant. Once the user has responded to all of the given situations, *Outcome-Diff: Questions* presents the percentage of situations for which each variant would have an acceptable outcome. A variant that satisfies more situations is more likely to match the user's intent. In the case of Figure 6, the interface shows that Program B matches a selected outcome in all situations where outcomes differ, while Program A matches none of them.

4.3 Property Differences

Sometimes, the user might care mainly about general trends or guarantees, such as whether the door will always be locked when Alice is asleep, as opposed to specific behaviors in specific situations. It is difficult for users to extract high-level trends from any interface presented so far, especially when variants have many differences.

***Property-Diff* helps users reason about broader behaviors in their automated systems by contrasting high-level properties held by the two variants** (Figure 7). In particular, our interface highlights *safety properties*, which are informally defined as statements indicating that "nothing bad happens in the system's execution" [3]. For instance, all three programs in Figure 2 have the safety property that "the front door will always be locked when Alice is asleep and it is nighttime." We designed this interface to be useful when differences in low-level rules or outcomes are unimportant and may even be burdensome to users.

The properties *Property-Diff* presents are based on templates from Zhang et al. [50] and are listed in Table 1. We excluded templates they found to confuse users, such as "[$state_1, ..., state_n$] will [*always*] be active together." We also condensed equivalent templates and separated those whose meaning can differ based on the number of device variables. Finally, we excluded timing-related properties, such as "[$state$] will [*never*] be active for more than

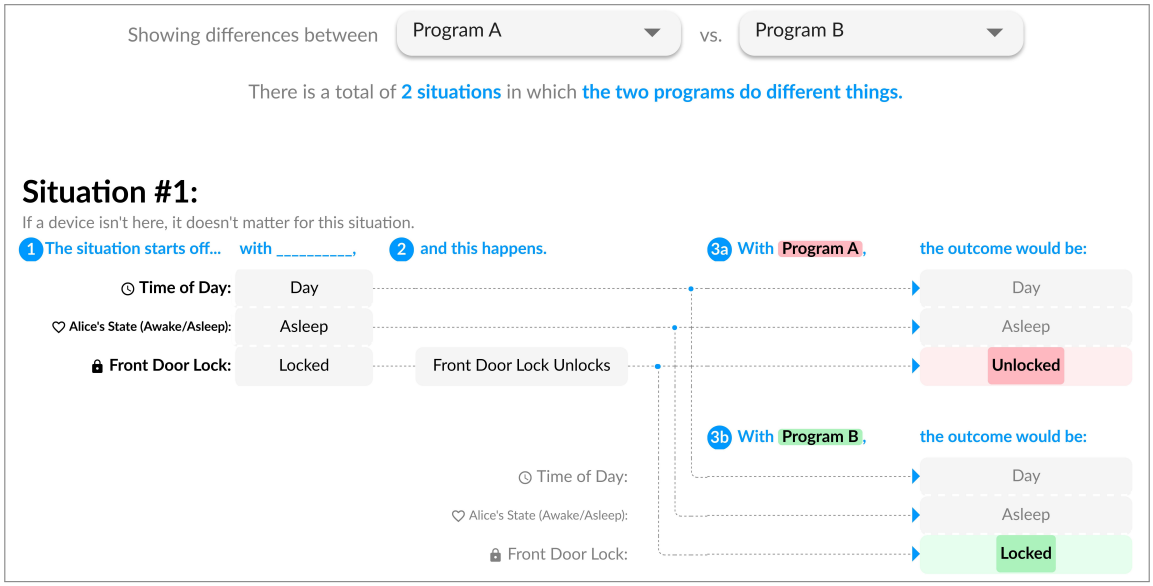
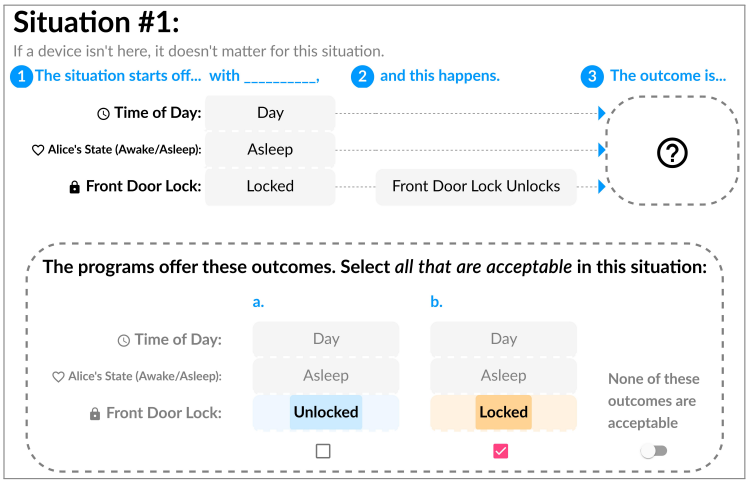
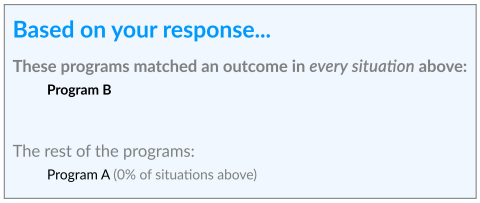


Figure 5: *Outcome-Diff: Flowcharts* showing a situation in which Programs A and B from Figure 2 produce different outcomes.



(a) The first situation *Outcome-Diff: Questions* shows with Programs A and B from Figure 2 as the variants. Users select zero or more desired outcomes.



(b) The results of *Outcome-Diff: Questions* appears when the user has made a choice for every situation. Note that for Programs A and B, there are two such situations. These example results occur when the user chooses to have the front door locked in both the first situation (Figure 6a) and the second situation.

Figure 6: Snippet of *Outcome-Diff: Questions* with Programs A and B from Figure 2 as variants.

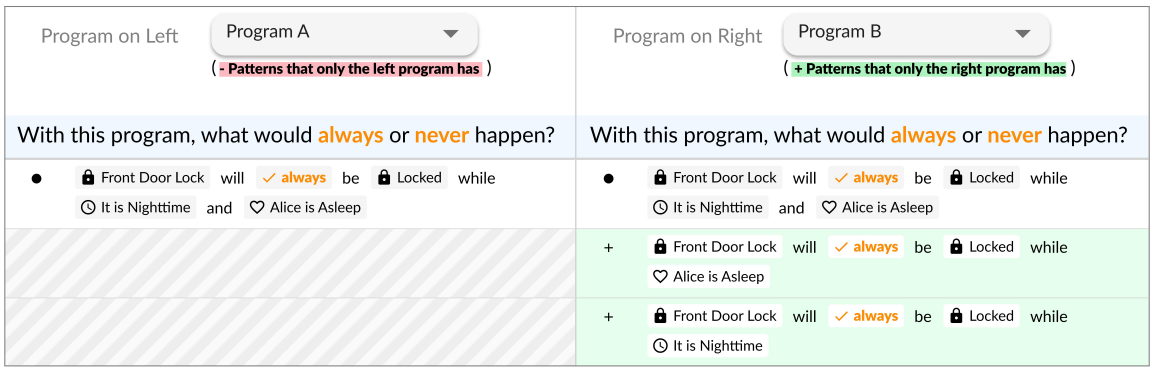


Figure 7: *Property-Diff* comparing the safety properties of Programs A and B from Figure 2.

[*duration*]," because they require different transition-system analyses than properties unrelated to timing (see Section 5 and [50]).

To maximize consistency across interfaces, we followed the layout and visual elements of *Text-Diff* for contrasting properties. Pilot participants did not understand the phrase "safety property," so we listed them as "patterns that are always or never true about the smart home." In addition to the properties that differ, *Property-Diff* also shows properties held by both variants. Pilot study participants preferred seeing all properties to gauge more comprehensively whether any variant satisfied the user study tasks. Unlike *Text-Diff*, we did not define how two different properties can be similar. Doing so requires more careful consideration of what makes two properties "similar," which we leave to future work.

5 ALGORITHMS FOR COMPUTING DIFFS

In this section, we present the algorithms underpinning the interfaces that we presented in Section 4. Section 5.1 shows how we compare text differences across variants for *Text-Diff*. Section 5.2 introduces transitions systems, which we use to represent TAP variants in a home. They serve as the basis for our algorithms for semantic-diff interfaces. Section 5.3 explains our algorithms to identify outcome differences across variants for *Outcome-Diff: Flowcharts* and *Outcome-Diff: Questions*. Section 5.4 introduces how we generate property differences across variants for *Property-Diff*.

5.1 *Text-Diff*

The main task in populating the *Text-Diff* interface algorithmically is to identify potential "modified" rules, which are pairs of rules from the two variants that are most similar to each other. The set of candidates is the Cartesian product of the two sets of rules from the two variants. Per our definition of "modified" rules in Section 4, we eliminate all candidates in which the rules differ by more than one subpart. We calculate the differences within each remaining pair of rules to be the number of conditions in which the pair differs, or 1 if the pair differs in their triggers or their actions (never both). We then choose the largest unique set of rule pairs with the smallest differences to be the set of "modified" rules.

5.2 Modeling the Home as a Transition System

All of our semantic-diff interfaces require reasoning about how particular TAP variants differ in behavior. To this end, we model each TAP variant and the smart home as a transition system [3, 50]. Each transition system incorporates rules of the variant as well as the subset of the home's devices (and their possible states) and environmental factors (e.g., weather, light levels, time of day) that affect or are triggered by these rules.

A node in the transition system represents every device and every environmental factor from the variant being in a particular state. Recall that each device or environmental factor in the home is represented by a variable. The set of states (nodes) in a given transition system is thus the Cartesian product of the set of states for each variable. For example, a system with n binary variables would have 2^n states. While a home with many devices and relevant environmental factors would seemingly require a huge transition system to model, the transition system actually only needs to include the devices and environmental factors directly related to the rules in the TAP variants being compared. For instance, if no rule in the TAP variants being compared is triggered by, nor acts upon, the home's thermostat, the thermostat can be excluded entirely from the transition system. Furthermore, a variable with a large number of possible states (e.g., the home's temperature) can be discretized. For instance, if the only aspect of temperature relevant to any rule in the variants being compared is whether or not the home is under 65°F, temperature can be treated as a binary variable. As a result, even for a home with many devices, the transition system typically remains small. For our user study (Section 6), the largest system—Task 6 (Abstraction)—had 7 binary variables and thus 128 states (nodes).

Transitions (edges) in the transition system reflect events that change one or more variables, which intuitively means that one or more devices or environmental factors has changed its state. Initially, these edges will represent manual transitions (e.g., a light can change state if someone manually turns it on) and natural environmental transitions (e.g., day will turn to night). When there are no automation rules, every pair of smart home states that differ in the state of exactly *one* variable will typically be connected by two transitions, one from each state to the other. However, this pattern does not hold true if not all transitions for a device are valid. For instance, in modeling a device that must always temporarily be in a "warming up" state before it reaches the "on" state, there will not be an edge from "off" to "on" nodes. As with Zhang et al. [50], we require that the valid transitions for a given type of variable be pre-specified as part of an overall model of a smart home.

Rules redirect transitions in the graph. Specifically, the in-transition of a state representing a particular rule's triggering event and conditions will be redirected to the state reflecting the eventual outcome of the action specified by the rule.

To further illustrate transition systems, we partially model our running example from Section 1. The states of this transition system will be the Cartesian product of whether Alice is asleep and whether the front door is locked (Figure 8), resulting in four states:

- $(\text{Alice}_{\text{awake}}, \text{door}_{\text{unlocked}})$
- $(\text{Alice}_{\text{awake}}, \text{door}_{\text{locked}})$
- $(\text{Alice}_{\text{asleep}}, \text{door}_{\text{unlocked}})$
- $(\text{Alice}_{\text{asleep}}, \text{door}_{\text{locked}})$

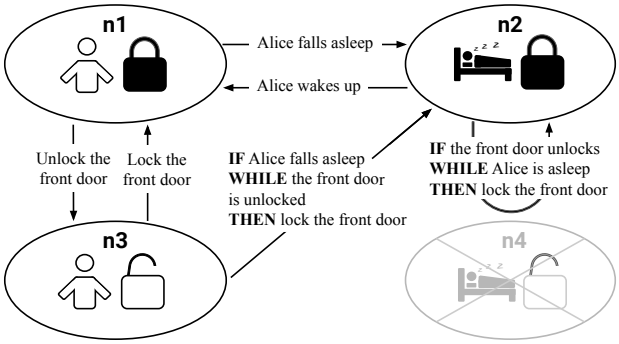
If Alice wakes up, which is a variable state change, the system transitions from $\text{Alice}_{\text{asleep}}$ to $\text{Alice}_{\text{awake}}$. If someone manually locks the door, the system transitions from $\text{door}_{\text{unlocked}}$ to $\text{door}_{\text{locked}}$.

Again, rules redirect these transitions. For instance, if Alice is awake with the door unlocked and then falls asleep, without any rules the system would move from state $n3$ to state $n4$ of Figure 8a. However, the rule "IF Alice falls asleep WHILE the front door is unlocked THEN lock the front door" instead redirects this edge to node $n2$ because the rule's action is to lock the front door.

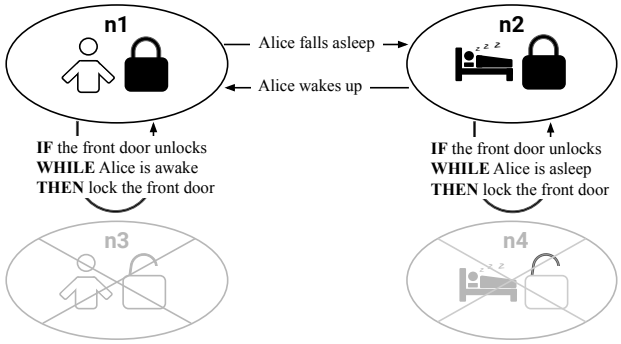
Our analyses for *Outcome-Diff* and *Property-Diff* rely on reachability analysis, using breadth-first search to identify reachable states in the system. This search process requires an initial state from which to start. Inadvertently choosing an invalid (unreachable) state as the starting point would result in the reachability analysis being incorrect, mistaking some unreachable states as reachable and vice versa. Most intuitively, real-world systems can choose the current state of the home as this initial state. In our user study,

Table 1: Properties that *Property-Diff* supports. The terms [*device_state*] and [*device_event*] refer to a device variable's state or event, while [*external_state*] refers to the state of a variable that cannot be controlled, like attributes of users and environmental factors. A [*state*] can be the state of any variable.

Property Template Based on AutoTap [50]	Example Phrasing in the *Property-Diff* Interface
[*device_state*] will [*always/never*] be active.	[*The lights*] will [*always*] be [*on*].
[*device_state*] will [*always/never*] be active while [$external\_state_1, \cdots, external\_state_n$]	[*The lights*] will [*always*] be [*on*] when [*it is nighttime*] and [*Alice is awake*].
[$state_1, \cdots, state_n$] will [*never*] be active together.	The smart home will [*never*] have [*the lights on*] and [*the window curtains open*] at the same time.
[*device_event*] will [*only/never*] happen when [$state_1, ..., state_n$].	[*The lights*] will [*only*] [*turn on*] when [*it is nighttime*].



(a) TS_1, a transition system for a trigger-action program consisting of two rules: "IF Alice falls asleep WHILE the front door is unlocked THEN lock the front door" and "IF the front door unlocks WHILE Alice is asleep THEN lock the front door." The first rule redirects transition $n3 \rightarrow n4$ to $n3 \rightarrow n2$. The second rule redirects the transition $n2 \rightarrow n4$ to $n2 \rightarrow n2$ (a self-transition). Assuming $n1$ is the initial state, this program ensures that $n4$ is unreachable.



(b) TS_2, a transition system for a trigger-action program consisting of two rules: "IF the front door unlocks WHILE Alice is awake THEN lock the front door" and "IF the front door unlocks WHILE Alice is asleep THEN lock the front door." Assuming $n1$ is the initial state, this program ensures both $n3$ and $n4$ are unreachable.

Figure 8: Transition systems (TS_1 and TS_2) of two TAP variants, both focusing on whether Alice is awake and whether the front door is locked. The system states are labeled $n1$ through $n4$. Arrows represent valid transitions. Alice is awake in $n1$ and $n3$, and asleep in $n2$ and $n4$. The front door is locked in $n1$ and $n2$, and unlocked in $n3$ and $n4$. If a rule triggers (redirects) a transition, we label the corresponding transition arrow with the rule. Unreachable nodes are colored gray and crossed out.

we choose a state that satisfies the goal of the task as the initial state. For example, if the goal is to make sure the door is always locked while Alice is asleep, we define the initial state to be any state except for the door being unlocked while Alice is asleep.

5.3 Outcome-Diffs

To identify all situations in which variants produce different outcomes, we compare their transition systems. Each variant is represented by its own transition system. The key intuition for detecting differences in outcomes is to compare the out-transitions for a given (corresponding) state across variants. If a given (corresponding) transition from this state differs across program variants, in that its destination state does not correspond across variants, that means the outcome of that event is different.

The *Outcome-Diff* interfaces share similar algorithms, so we detail only the *Questions* algorithm for n variants, presenting its pseudocode as Algorithm 1. *Flowcharts* uses this algorithm for $n = 2$.

First, we generate the transition systems $TS_1 \cdots TS_n$ for the n variants. We then find the nodes that are reachable in every transition system, which are the system *states* that are possible with all of the variants. For all outgoing edges of these nodes (i.e., every possible *event* happening from these *states*), we identify the actions they would trigger with $Variant_1 \cdots Variant_n$. For each situation, we group the variants based on the actions they take, combining groups for which the actions lead to identical outcomes. We also merge the situations to minimize redundancy. We then convert the remaining situations to multiple-choice questions, each asking "what should the program do when *event* happens in *state*," with the choices being all possible outcomes from $Variant_1 \cdots Variant_n$.

We present an example for $n = 2$ (identical to *Outcome-Diff: Flowcharts*) with Figure 8. The first program consists of two rules: "IF Alice falls asleep WHILE the front door is unlocked THEN lock the front door" and "IF the front door unlocks WHILE Alice is asleep THEN lock the front door." The second program consists of two rules as well: "IF the front door unlocks WHILE Alice is awake THEN lock the front door" and "IF the front door unlocks WHILE Alice is asleep THEN lock the front door." Assuming $n1$ is the initial state, the two respective transition systems share two possible *states*: $n1$ and $n2$.

Input : $Variant_1, \cdots, Variant_n$
Output: *situationDiffs*, a list of (*startState*, *event*, *behaviorMap*) where variants behave differently by taking different actions. *behaviorMap* is a map from behaviors to variant ids.
Function *findSituationDiffs*($Variant_1, \cdots, Variant_n$)
  $TS_1 \cdots TS_n$:= transition systems of smart home implementing $Variant_1 \cdots Variant_n$;
  $R_1 \cdots R_n$:= sets of reachable states in $TS_1 \cdots TS_n$;
  $R := R_1 \cap \cdots \cap R_n$;
  situationDiffs := $\emptyset$;
  for *Every state in R* **do**
    for *Every event that can happen from state* **do**
      $beh_1 \cdots beh_n$:= Actions triggered by $Variant_1 \cdots Variant_n$ when *event* happens under *state*;
      if $beh_1, \cdots beh_n$ *are not all the same* **then**
        behaviorMap := {} (empty map);
        for i *in* $1 \cdots n$ **do**
          if beh_i *is not in behaviorMap* **then**
            $behaviorMap[beh_i] := \emptyset$;
          $behaviorMap[beh_i] := behaviorMap[beh_i] \cup \{i\}$;
        situationDiffs := *situationDiffs* $\cup$ {(*state*, *event*, *behaviorMap*)};
  return *situationDiffs*

Algorithm 1: Pseudocode for the *Outcome-Diff: Questions* algorithm. We identify situations (a system state and an event) that result in any of the n variants producing different outcomes. *Outcome-Diff: Flowcharts* uses this algorithm for $n = 2$.

From *state* $n1$, the possible events are Alice falling asleep and the door becoming unlocked (by someone or another rule). In the left transition system, when the front door becomes unlocked while Alice is awake at $n1$, the system allows this transition to $n3$. In the right transition system, the system would instead transition back to $n1$ with Alice awake and the door still locked. For *Outcome-Diff: Questions*, the algorithm would generate this question (in flowchart form): "The situation starts off with Alice awake and the front door locked. Now the front door unlocks. What should the outcome of this situation be?" The choices would be "have Alice awake and the front door unlocked" and "have Alice awake and the front door locked." In *Outcome-Diff: Flowcharts*, a flowchart would show this situation with the two choices as diverging outcomes (Section 4.2).

When there are multiple conditions or actions, we combine situations that only differ in conditions that do not affect the outcome. For example, if two situations are identical in conditions and in the set of outcomes, except that Alice is asleep in one situation and awake in the other, we combine them into a single situation and do not show whether Alice is asleep on the interface.

5.4 *Property-Diff*

Our algorithm underlying *Property-Diff* centers on analyzing the unreachable nodes in variants' transition systems. The key intuition is that the states of a given variable or given combination of variables that are always unreachable can be directly mapped to human-intelligible safety properties for the *Property-Diff* interface.

We first extract safety properties from each variant. We then separate the safety properties shared by both variants. The safety properties we consider (Table 1) consist of properties based on states (**S-properties**, such as "the door will never be unlocked when Alice is asleep") and properties based on events (**E-properties**, such as "the door will never *unlock* when Alice is asleep"). Algorithm 2 presents our pseudocode. We find S- and E-properties separately.

Our algorithm considers that safety properties can be stated as logical expressions, and therefore multiple logical expressions are equivalent. For example, the S-property "the front door will always be locked" implies other S-properties, such as "the front door will always be locked *while Alice is awake*" and "the front door will always be locked *while Alice is asleep*." These latter two properties can combine to yield the first by plugging the variable states into a Boolean expression in disjunctive normal form—"*(Front door being locked AND Alice being awake) | (Front door being locked AND Alice being asleep)*"—and simplifying this expression. This approach has been used in related applications, such as simplifying smart building rulesets [41]. Note that variables with more than two states (e.g., light hue color) and range variables (e.g., temperature) require a few more steps, but can also be simplified in this manner. E-properties can also be written in the form of other E-properties and combined with logic minimization. Assuming that the front door is locked in the initial state, the first S-property can also be stated as the E-property "the front door will never *unlock*" and (implicitly) the E-property "the front door will never *lock*" as it is already locked.

Displaying a large number of properties would likely overwhelm users. Therefore, our algorithm extracts properties in their most simplified form, which in this case is the first S-property. In addition, because positive statements are typically easier to understand than negative ones, when possible we convert properties about states or events that can *never* occur into properties about states or events that will *always* occur. For example, "the front door will never be unlocked" is restated as "the front door will always be locked."

We first extract S-properties by identifying unreachable nodes in each system. An unreachable node indicates a corresponding state that can never hold true in that system. For example, if the node corresponding to $(\text{Alice}_{\text{asleep}}, \text{door}_{\text{unlocked}})$ is unreachable, then the property "the front door will never be unlocked while Alice is asleep" is true for the system. Instead of converting each node into a property in this manner, we perform logic minimization over the set of such nodes to merge them based on common variable states, and then generate corresponding S-properties. For example, if the node corresponding to $(\text{Alice}_{\text{awake}}, \text{door}_{\text{unlocked}})$ is unreachable in addition to the node we just mentioned, then we merge these two nodes to be $(\text{door}_{\text{unlocked}})$, yielding the property "the front door will never be unlocked." This approach lets us generate one general property, as opposed to two specific properties. Our implementation uses the SymPy library [29] for minimization, which relies on the Quine-McCluskey algorithm [27]. Other logic minimization algorithms would be valid as well. We then identify nodes unreachable in both systems and repeat the same process on them to identify S-properties shared by both systems. This step not only helps with the next step, in which we determine S-properties

unique to one variant but not the other, but it also outputs the shared S-properties that our interface shows in addition to unique S-properties. Finally, we subtract the shared S-properties from the S-properties of each transition system. Each system's remaining S-properties are unique to that system, and thus that variant.

As an example, we run our analysis on TS_1 and TS_2 in Figure 8. With $n1$ as the initial state, we find that $n4$ ($\text{Alice}_{\text{asleep}}$, $\text{door}_{\text{unlocked}}$) is unreachable in both TS_1 and TS_2. Therefore, the two systems both have the property "the front door will never be unlocked while Alice is asleep." This is also the only S-property held by TS_1. We find that both $n3$ ($\text{Alice}_{\text{awake}}$, $\text{door}_{\text{unlocked}}$) and $n4$ are unreachable in TS_2, which produces the property "the front door will always be locked." The shared S-property cannot be subtracted from this property. Therefore, we will tell the user that this property is unique to TS_2. Alternatively we could say that the property unique to TS_2 is "the front door will always be locked *while Alice is awake*," but we designed the algorithm to quickly deduce properties at the most abstract level and avoid outputting too many properties when a few would be equivalent or imply them.

For E-properties, we identify self-transitions and repeat the same algorithm as for S-properties. A self-transition indicates that an event can never occur, possibly under some situation (e.g., "the front door will never unlock when Alice is awake" based on the self-transition from $n1$ in TS_2, but not TS_1). Some E-properties are implied by S-properties because their corresponding self-transitions contribute to an unreachable node. Therefore, we ignore these self-transitions. This is the case for all self-transitions in TS_1 and TS_2 of Figure 8. Therefore, we do not show any E-properties for them.

5.5 Scalability

Although real-world systems may have large programs and transition systems, we expect users will want to compare subsets of rules related to a specific task (e.g., controlling the door lock) to related sets of rules, as described in Section 1. While our algorithms could become computationally intractable for a sufficiently large transition system, we believe transition systems for realistic tasks will generally be small from designing the study tasks. For example, in a realistic TAP diff task—Task 6 (Abstraction) with 7 binary attributes—it took only 8 seconds on a commodity laptop to generate results from scratch (showing 19 situations) for *Outcome-Diff: Flowcharts*, and 5 seconds for *Property-Diff*. Precomputation and caching would further reduce the time required to run the algorithms and help the approach scale further.

6 USER STUDY METHODOLOGY

To evaluate whether our interfaces could help users understand TAP variants, we performed a 107-participant online user study. We studied the following research questions:

- **RQ1:** Compared to *Rules* or *Text-Diff*, do semantic-diff interfaces equip users to more accurately choose the correct program variant(s) out of a set of prospective variants to match a motivating goal?
- **RQ2:** How does the relative performance of these interfaces compare across programs with different characteristics (e.g., length, complexity, number of prospective variants)?
- **RQ3:** Do specific interfaces help users reason about TAP program differences more (a) confidently and (b) quickly?

Input : TS_1, TS_2: transition systems of smart home implementing $Variant_1$, $Variant_2$
Output: $allProperties$: 3-tuple of properties unique to $Variant_1$, properties unique to $Variant_2$, and properties held by both variants
Function *findAllProperties(TS_1, TS_2)*

U_1 := set of nodes unreachable in TS_1;
U_2 := set of nodes unreachable in TS_2;
$U_{\text{both}} := U_1 \cap U_2$;
// S_{both}: S-properties shared by both variants
S_{both} := toSproperties(minimizeLogic(U_{both}));
// S_1, S_2: unique S-properties of each variant
S_1 := toSproperties(minimizeLogic(U_1)) $-S_{\text{both}}$;
S_2 := toSproperties(minimizeLogic(U_2)) $-S_{\text{both}}$;

T_1 := set of self-transitions from reachable nodes that do not contribute to U_1 in TS_1;
T_2 := set of self-transitions from reachable nodes that do not contribute to U_2 in TS_2;
$T_{\text{both}} := T_1 \cap T_2$;
// E_{both}: E-properties shared by both variants
E_{both} := toEproperties(minimizeLogic(T_{both}));
// E_1, E_2: unique E-properties of each variant
E_1 := toEproperties(minimizeLogic(T_1)) $-T_{\text{both}}$;
E_2 := toEproperties(minimizeLogic(T_2)) $-T_{\text{both}}$;

$uniqueProperties_1 := S_1 \cup E_1$;
$uniqueProperties_2 := S_2 \cup E_2$;
$properties_{\text{both}} := S_{\text{both}} \cup E_{\text{both}}$;
return
($uniqueProperties_1$, $uniqueProperties_2$, $properties_{\text{both}}$)

Algorithm 2: Pseudocode for the *Property-Diff* algorithm. We identify properties that are unique to each variant, as well as properties shared by both variants. "minimizeLogic()" plugs nodes or edges into a logic expression in disjunctive normal form, simplifying this expression. Self-transitions that contribute to unreachable nodes are those that, due to their redirection, helped make the original destination node unreachable.

To address these questions, we designed six program-comparison tasks modeled after potential scenarios in which the user encounters variants (Section 6.3). Each participant would complete these tasks in a randomized order using a randomly assigned interface. For each task, participants could choose to see the programs themselves (listed in prose) by clicking on a "program button." Our tutorial encouraged participants to focus on the interface, briefly mentioning that this "program button" feature was available. As described in Section 4, we piloted interface designs on users with various technical backgrounds before the study to refine our designs.

6.1 Recruitment

We recruited participants with Prolific Academic [37], a recommended alternative [40] to other recruitment platforms like Amazon Mechanical Turk. Participants had to be 18 years or older with US nationality to take part in the study. We also required them to

have a 90% or higher approval rate from at least 10 previous submissions. As we expected our interfaces to be particularly helpful for non-technical users, our recruitment description emphasized that we did *not* require experience with programming or smart homes. We asked participants to take the study on a computer, not a phone.

6.2 Study Overview

We first eased the participants into TAP and their randomly assigned interface. After consenting to the study, participants read a one-page description of trigger-action programs. They then completed an interactive tutorial of their assigned interface on a sample task. If a participant failed two of the three simple attention check items, the study terminated early. Otherwise, participants continued onto the main portion of the study for their interface.

In the main portion, participants completed the six program-comparison tasks in a randomized order. For each task, we recorded participants' responses and durations. After each task, we asked participants to rate on a 7-point Likert scale how much they agreed with the following three statements: "I am confident that my answer is correct"; "I found the task mentally demanding"; and "I found the interface helpful in completing the task." We also asked them to briefly describe their approach to completing the task and how the interface helped or hindered.

Following the six tasks, we collected more in-depth information about participants' experiences using the interfaces, as well as relevant background that could have influenced their performance and experience. We asked participants to describe their general approach more specifically, including the parts of the interface that were most helpful, least helpful, and most confusing. To illustrate the level of detail we were looking for in their descriptions, we provided an example describing a hypothetical approach for accepting Facebook friend requests. We also asked them whether they relied mostly on the interface itself, the programs, or both. We quantitatively gauged interface usability with the 7-point version of the System Usability Scale (SUS). The study concluded with questions about the participant's demographics and technology background.

This study was approved by the UChicago IRB. We compensated each participant $10.00. The average completion time was slightly over an hour. We include the full survey instrument and the text of the tasks in our online supplementary materials [53].

6.3 Task Design

To understand how our interfaces can help users, we designed the tasks to emulate scenarios in which users need to compare variants (Table 2 and Table 3). We included at least one task for each interface, including the controls, that we expected to highlight the unique advantages of the interface. Each task asked the participant to compare between two or more programs. In all tasks, we evaluated whether participants could identify differences across variants that the task requested. Our tasks followed these hypothetical scenarios:

(1) Task 1 (Straightforward): Given an original program, a desired behavior extension, and modified versions of this program, could the participant determine which of the modified versions maintained the original program behaviors, but extended them as specified?

Table 2: Overview of the complexities of the tasks.

Task	Complex Programs	Situations with Different Outcomes	Differ in Properties	Many Variants
1 - Straightforward		✓		
2 - Simple Logic		✓		
3 - Redundant Programs	✓	✓	✓	
4 - Hidden Similarity	✓	✓	✓	
5 - 27 Variants		✓	✓	✓
6 - Abstraction	✓	✓	✓	

(2) Task 2 (Simple Logic) and Task 6 (Abstraction): Given an original program, some observations about its undesirable behaviors, and modified versions of this program, could the participant determine which of the modified versions would exhibit the desirable behaviors instead?
(3) Task 3 (Redundant Programs) and Task 4 (Hidden Similarity): Given an original program, its goal, and modified versions of this program, could the participant determine which of the modified versions met the same goal as the original program?
(4) Task 5 (27 Variants): Given multiple programs and a set of goals, could the participant determine which of the programs meet all of the goals?

We designed the tasks with the following characteristics to best highlight each interface. In general, we expected *Outcome-Diff* interfaces to outperform the controls on tasks with complex programs. We defined complex programs to be long programs with many rules affecting the same variables, which may hinder participants from tracking all behavior differences. We arbitrarily decided that a long program would contain at least six rules, with some rules containing three or more conditions. Meanwhile, we expected participants using the control interfaces to do well on tasks with variants that were not complex. Therefore, we designed Tasks 3 (Redundant Programs), 4 (Hidden Similarity), and 6 (Abstraction) to have complex programs, and Tasks 1 (Straightforward) and 2 (Simple Logic) to have simple programs. Variants in the former three tasks also differed in properties, for which *Property-Diff* should help. To see whether participants would benefit from the form-based interaction of *Outcome-Diff: Questions*, we designed Task 5 (27 Variants) to have many variants. In all tasks, variants had situations with different outcomes, for which we expected *Outcome-Diff* participants to do well even if the variants were not long enough to be complex.

6.4 Statistical Analyses

Our statistical analysis approach depended on whether our variables of interest were categorical or numeric. For both cases, we first conducted an omnibus test to determine whether the variable of interest varied by interface. For significant omnibus results (using $\alpha = 0.05$), we then conducted pairwise comparisons across all interfaces to determine which interfaces varied significantly from which other interfaces. We were particularly interested in how our *Outcome-Diff* and *Property-Diff* interfaces compared to the two control conditions, which reflected existing interfaces. Thus, we report the results of these comparisons most prominently. To minimize the possibility of Type I errors due to multiple testing, we corrected p-values within each family of omnibus tests and within each set of pairwise comparisons using the Holm method.

Table 3: Detailed summary of the tasks.

1 - Straightforward	Given an original program and a modified version, the participant decides whether the modified version does exactly what the original program does, except it also turns off the faucet when Alice leaves the bathroom.
2 - Simple Logic	Given an original program and two modified versions, the participant decides which of the two versions (if any) would turn off the AC when neither Alice nor Bobbie is at home, but does not affect the AC if either one of them is home.
3 - Redundant Programs	Given an original program that meets a specified goal and a modified version, the participant decides whether the modified version also meets the specified goal. The goal is to secure the home by keeping the security camera on when the front door is unlocked or Alice is asleep. Both programs contain redundant rules.
4 - Hidden Similarity	Given an original program that meets a specified goal and two modified versions, the participant decides which of the two versions (if any) would meet the specified goal. The goal is to keep exactly one of three windows open in the house at any given time.
5 - 27 Variants	Given 27 programs that meet the same goal, the participant decides which of the programs (if any) would meet a second goal. The first goal is to ensure that the living room window being open, the TV being on, and the Roomba being on will never occur simultaneously. The second goal is to ensure that Alice gets to enjoy fresh air from the living room window without disruption to her TV time.
6 - Abstraction	Given an original program that fails to meet a specified goal and two modified versions, the participant decides which of the two versions (if any) can meet the goal. The goal is to save electricity by ensuring that only one of eight smart devices (AC, coffee pot, lights, TV, speakers, and the Roomba) is on at any given time.

For both omnibus and pairwise comparisons between interfaces for categorical data, we used Fisher's Exact Test. We chose Fisher's Exact Test instead of the more familiar Pearson's χ^2 test because the latter is considered unreliable when expected cell counts are smaller than 5, which was often the case in our data. These categorical comparisons analyzed how the correctness of participants' answers differed across interfaces for each of the six tasks, as well as how the use of the program button varied across interfaces for each of the six tasks. Thus, each instance of Fisher's Exact Test was run on a 2×5 contingency table (binary outcomes across five interfaces).

For omnibus comparisons of numeric data, we used the Kruskal-Wallis H test. For pairwise comparisons, we used its analogue for 2 groups, the unpaired Wilcoxon rank-sum test, also known as the Mann-Whitney U test. These non-parametric tests are appropriate for data that is not normally distributed and for ordinal data, which was the case for our quantitative data. In particular, we analyzed how the number of tasks participants answered correctly, the number of tasks for which participants used the program button, System Usability Scale scores, and total time taken (summed across tasks) varied by interface. We also tested whether participants' Likert-scale responses to the following three prompts (averaged across tasks) varied by interface: their confidence in their answers to the tasks, whether they considered the tasks demanding, and whether they considered the interface helpful in completing the tasks.

We also created regression models to understand how both the assigned interface and numerous demographic characteristics correlated with the number of tasks each participant answered correctly and the number of tasks for which they used the program button. Because both factors are counts, we created Poisson regression models. The independent variables (IVs) were the assigned interface, number of tutorial questions answered correctly, and the participant's age, gender, education level, computer science background, familiarity with IFTTT, and ownership of IoT devices. For categorical variables with many categories, we binned similar categories. Using the same IVs, we also created generalized linear models for the total time a participant took across the six tasks and their average confidence rating across tasks, both of which we treated as continuous. For all regressions, we created (and report) a parsimonious model developed through backward selection by Akaike Information Criterion (AIC). Our supplementary materials [53] provides the full regression tables. For brevity, we report only p-values in the body of the paper.

6.5 Limitations

Our study required about an hour of our participants' time, with no expectation of smart home programming experience. Therefore, we cannot make conclusions about these interfaces regarding long-term smart home use. As we mention in Section 7, although we emphasized that no experiences with programming nor smart homes was required, many of our participants had programmed before and/or owned IoT devices. Experimenter bias is also possible, but we mitigate this concern by measuring whether participants completed the tasks correctly.

7 USER STUDY RESULTS

In this section, we report the results of our user study. Overall, *Rules* worked well for the simple tasks, as did most other interfaces. For the complex tasks, however, *Rules* participants performed poorly while participants using semantic-diff interfaces performed significantly better. The latter participants also found the complex tasks significantly less demanding than participants using *Rules*.

7.1 Participants

We received 124 complete responses. We excluded both responses from one participant who did the study twice (and also failed the attention check), seven who gave low-effort or nonsensical answers to the text responses, four who copied and pasted their answers throughout the study, and four who did not complete the tutorial. After applying these criteria, 107 responses remained for analysis.

All participants were between 18 and 74 years old, with 50.5% in the 25–34 range and 28.0% in the 18–24 range. 51.4% of participants identified as women, 46.7% as men, and 1.9% as non-binary. Regarding education, 33.3% held a 4-year college degree, 20.6% had some college background, 15.9% were high school graduates, 11.2% held a 2-year college degree, and 0.9% held a graduate degree. 56.1% of participants had some technical background; they had programmed before, completed a CS-related class, and/or had a CS-related degree or job. 81.3% of participants were not familiar with the IFTTT service [19], but 66.4% of participants owned at least one IoT device.

7.2 Correctness

On tasks requiring comparisons of complex variants, we found that participants using the novel semantic-diff interfaces generally outperformed those using *Rules* and *Text-Diff* interfaces (Figure 9).

Even though participants assigned our novel interfaces completed more tasks correctly than those assigned the control interfaces, the time participants took to complete the tasks did not differ significantly across interfaces. In other words, in the same amount of time, participants were more successful completing the tasks using the semantic-diff interfaces than the control interfaces.

Table 4 compares the semantic interfaces and control interfaces by task. In general, *Outcome-Diff: Flowcharts* helped participants identify differences in outcomes when comparing between a few variants and when the variants differed in only a few situations, as in Task 3 (Redundant Programs) and Task 4 (Hidden Similarity). When the number of variants or situations became large, as in Task 5 (27 Variants) and Task 6 (Abstraction), *Outcome-Diff: Flowcharts* was less successful. *Outcome-Diff: Questions* was helpful when choosing between few variants, as in Task 3 (Redundant Programs) and Task 4 (Hidden Similarity). It was also helpful even when the variants differed in many situations with many variables, as in Task 6 (Abstraction). For that task, all variables had the same attributes (being on/off) and the goal was abstract enough that it did not matter exactly which variable's state differed. *Property-Diff* was helpful when there were abstract differences across a few variants, as in Task 3 (Redundant Programs) and Task 6 (Abstraction), but not if there were many variants, as in Task 5 (27 Variants), or if the variants had many properties, as in Task 4 (Hidden Similarity).

Compared to the other interfaces, we expected that *Rules* would only work well for Task 1 (Straightforward) and Task 2 (Simple Logic) because they involve short and simple variants. *Rules* participants were significantly more likely to complete the first task correctly than *Property-Diff* participants. As Table 4 shows, though, *Rules* did not outperform the semantic-diff interfaces in any other task. *Rules* participants found this first task easy, including P52: "*…it made it easy to notice the different modification that was being added to the program.*" A few, however, mentioned that it would be more convenient to see the variants side-by-side. *Property-Diff* participants mostly relied on the "program button" feature (viewing the rules themselves) rather than their assigned interface. This decision is logical because there were no safety properties that could be derived from either variant for the interface to show. While the interface explicitly stated that there were no patterns to show, some participants nonetheless thought the interface was buggy or "*unavailable*" (P61). A few others, such as P43, mistakenly believed that the lack of comparison meant "*the programs were the same.*"

For Task 2 (Simple Logic), we did not find significant differences in correctness between any semantic-diff interface and any control interface. *Rules* participants liked the interface because "*the layout was clear and easy to understand*" (P93). *Text-Diff* participants, like P51, appreciated having the differences highlighted: "*It's great helping me compare programs because it is color coordinated and they are side by side.*" While some *Outcome-Diff: Flowcharts* participants found the interface useful, a few felt ambivalent, like P30: "*The interface allowed me to see the outcomes combined together. Its design did not really help nor hinder my ability.*" Some thought it needed more information, such as for "*separating out what the original program does entirely from what the modified program does*" (P89). On the other hand, *Outcome-Diff: Questions* participants largely found the interface straightforward and felt it "*helped [them] visualize the scenario*" (P72). *Property-Diff* participants found the interface simple, although a few mistook the properties for the actual rules.

In Task 3 (Redundant Programs), with the exception of *Property-Diff* against *Text-Diff*, participants using the novel interfaces significantly outperformed those using the control interfaces (Table 4). Most *Outcome-Diff: Flowcharts* participants found their interface helpful. P26 stated, "*The interface helped, as there were less variables. I could focus on the one situation and identify faults with the second program as there were only two to compare.*" *Outcome-Diff: Questions* participants mostly found their interface straightforward as well, such as P7: "*I think visually showing all the combination was helpful, so I could see easily if the program met them.*" *Property-Diff* participants felt similarly. P48 wrote, "*The interface made it clear that the rules that were desired were only partially met and the interface made this fairly simple to determine.*" A few *Rules* and *Text-Diff* participants found the large number of rules in this task frustrating, but most still found their interface straightforward. P99 found *Rules* "*well structured with a good layout which makes it easy and possible to determine the comparison between both programs.*" For *Text-Diff*, P3 wrote, "*There were a lot of steps to sort through, but the color coding that told me when something was different helped.*"

For Task 4 (Hidden Similarity), *Outcome-Diff: Flowcharts* performed significantly better than the controls, and *Outcome-Diff: Questions* performed significantly better than *Rules*, as shown in Table 4. *Outcome-Diff: Flowcharts* participants found the interface mostly straightforward, albeit with a few hurdles. P5 wrote, "*I was able to refer to final configuration of open and closed windows though I had trouble following the "if this situation happens" aspect of it.*" P21 said, "*It was a bit technical but I think I was able to get it.*" *Outcome-Diff: Questions* participants felt confident, such as P6: "*The interface helped significantly, having % match from previous situations let me quickly look to the options and see if any fit the suggested change.*" *Rules* participants found the interface "*a little overloaded*" (P27) or "*cluttered*" (P31). P24 explained, "*I couldn't see all options at the exact same time to compare them all. I had to memorize and try to figure out the differences.*" *Text-Diff* participants liked the diff highlights, but thought the tasks made the interface confusing. P14 explained, "*It made it very difficult to concentrate and be able to determine the correct answer. Having three windows' statuses being repeated over and over also made it confusing to read.*"

We did not observe significant differences between semantic-diff participants and control participants for Task 5 (27 Variants). However, we found that *Outcome-Diff: Questions* participants outperformed *Outcome-Diff: Flowcharts* participants. Many *Outcome-Diff: Questions* participants, like P7, found the task relatively easy: "*It made it very quick to eliminate options that didn't meet the requirements.*" On the other hand, *Outcome-Diff: Flowcharts* did not reduce the mental load of the task for its participants. P17 stated, "*I had difficulty aligning how the interface views aligned with the requirements. It was not obvious or intuitive to me.*" *Rules* participants, like P27, felt ambivalent about the interface: "*The interface was fine, but it didn't really help to find the result quickly, since I had to click on each program in the dropdown.*" *Text-Diff* participants felt similarly, like P14: "*It was hard to scroll through all 27 programs, and it was a tedious and long process, but simple to understand and visually evaluate.*" The same was true for *Property-Diff* participants, with P96 explaining, "*This one was fairly easy to figure out and the design*

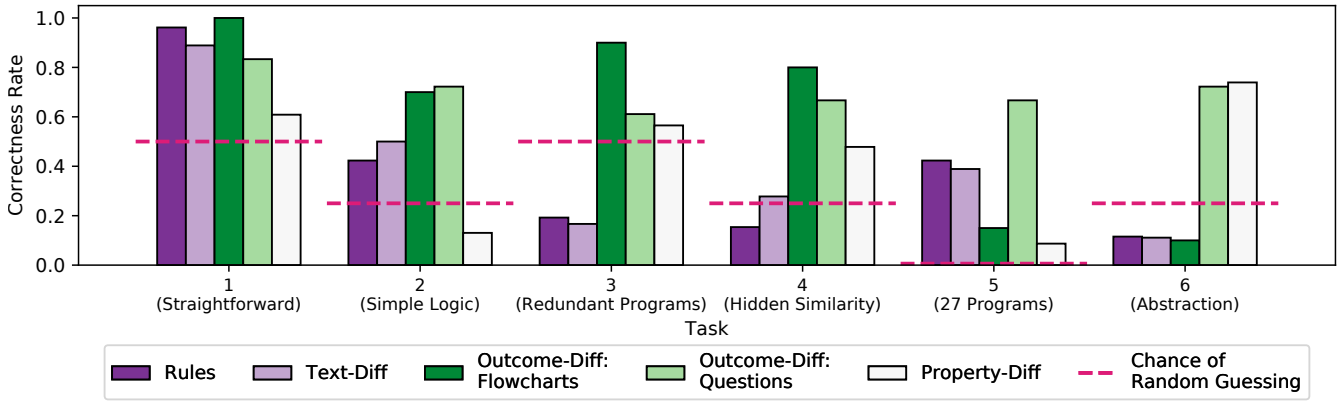


Figure 9: Participants' correctness per interface and task. Dashed lines represent chance of random guessing.

Table 4: Differences in task correctness between novel interfaces (columns) and controls (rows). The table shows whether a significantly larger (✓) or smaller (×) fraction of participants who used a novel interface answered the task correctly compared to those who used a control. P-values below task names are omnibus tests (Fisher's Exact Test) across all interfaces, while those in table cells are pairwise comparisons. All p-values were corrected for multiple testing using the Holm method.

		Outcome-Diff: Flowcharts	*Outcome-Diff: Questions*	*Property-Diff*
1 - Straightforward	*Rules*			× $p = 0.028$
($p = 0.001$)	*Text-Diff*			
2 - Simple Logic	*Rules*			
($p = 0.001$)	*Text-Diff*			
3 - Redundant Programs	*Rules*	✓ $p < 0.001$	✓ $p = 0.040$	✓ $p = 0.049$
($p < 0.001$)	*Text-Diff*	✓ $p < 0.001$	✓ $p = 0.049$	
4 - Hidden Similarity	*Rules*	✓ $p < 0.001$	✓ $p = 0.009$	
($p < 0.001$)	*Text-Diff*	✓ $p = 0.029$		
5 - 27 Variants	*Rules*			
($p = 0.001$)	*Text-Diff*			
6 - Abstraction	*Rules*		✓ $p < 0.001$	✓ $p < 0.001$
($p < 0.001$)	*Text-Diff*		✓ $p = 0.004$	✓ $p = 0.001$

of the interface did help explain things. You did however have to be careful and watch for what was stated."

For Task 6 (Abstraction), *Property-Diff* and *Outcome-Diff: Questions* participants performed significantly better than the control groups (Table 4). *Property-Diff* participants mostly thought the interface made the task easy, such as P90: "*It told me exactly yes or no if the program would do what was needed.*" Similar to Task 1 (Straightforward), however, a few participants were frustrated when *Property-Diff* did not show properties for a variant (as it lacked properties by our definition), like P69: "*It's frustrating to see that lack of the always or never rules and makes it feel like there's no program.*" Due to an experimental error, *Outcome-Diff: Questions* only showed participants 7 out of 21 questions. Nonetheless, the fact that they outperformed the control group raises interesting questions about the design of *Outcome-Diff: Questions*, as we discuss in Section 8. Some *Outcome-Diff: Questions* participants, like P7, felt that "*not having to keep track of which options were on and off in each situation made it very easy to quickly determine which combinations met the requirements.*" Some others, however, were overwhelmed: "*There were so many choices that it became cumbersome to figure out*" (P65). Most *Rules* participants found the interface "*okay*" (P76) or "*not very user friendly*" (P35). P24 explained, "*It was hard because of all the information that was presented and the only way to tell*

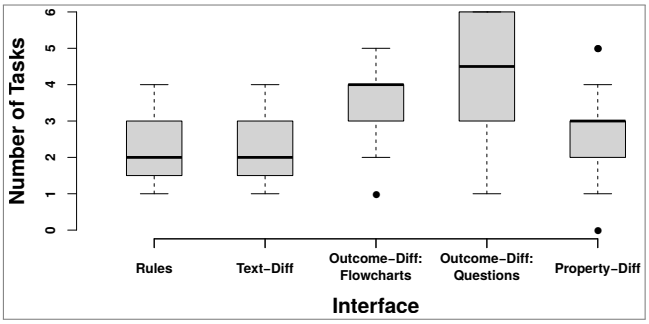


Figure 10: The distribution of the number of tasks a given participant answered correctly.

the difference between the programs was to either switch back and forth or memorize." Many *Text-Diff* participants found the layout to be overwhelming, such as P63: "*When there are too many requirements listed out separately in the "while" section, it can get a little overwhelming to keep it straight.*"

The tasks in this study were not randomly chosen. However, because they cover a variety of different comparison scenarios, we also compared how many tasks each group of participants answered

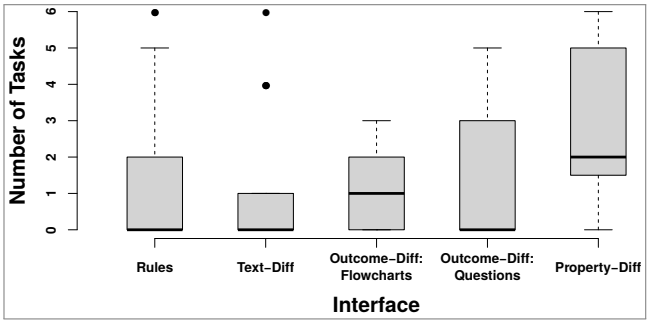


Figure 11: The distribution of the number of tasks for which a given participant used the program button.

correctly. On average, participants correctly answered 2.9 of the 6 tasks. Overall performance varied significantly between interfaces ($\chi^2(4) = 30.604$, $p < 0.001$), as shown in Figure 10. Participants using the two interfaces with low-level information, *Outcome-Diff: Flowcharts* and *Outcome-Diff: Questions*, completed significantly more tasks correctly than those with other interfaces (vs. *Rules*: $p < 0.001$ and $p = 0.010$, respectively; vs. *Text-Diff*: $p = 0.006$ for both; vs. *Property-Diff*: $p = 0.014$ and $p = 0.011$, respectively).

7.3 Reliance on Seeing Programs

On average, participants used the "program button" feature to see the rules themselves for 1.6 of the 6 tasks. The number of tasks for which participants chose to look at the programs differed significantly between interfaces ($\chi^2(4) = 12.024$, $p = 0.017$), but we did not find any significant pairwise differences (Figure 11). In our Poisson regression, we observed that *Property-Diff* participants were more likely to consult this feature than our baseline *Rules* participants ($p = 0.004$), while women were more likely to do so than men ($p = 0.007$). Furthermore, participants without any college education were more likely to consult this feature than participants with a graduate degree ($p = 0.049$).

Half of the *Rules* participants, who already saw the rules themselves in their interface, did not rely on this feature. Unexpectedly, though, some relied on both this feature and the interface to compare variants more easily or to show long programs on the same screen. A few relied only on this feature; one found its text easier to read than the three-column layout. Many *Text-Diff* participants relied on the interface alone because it was easier to use, but some found both the interface and the program button helpful.

Almost all *Outcome-Diff: Flowcharts* participants relied on the interface for the most part because they preferred seeing the outcome differences. P30 stated, "*It's easier to just focus on the outcomes that way and see whether it meets the conditions or not. Also, the interface is a visual compared to the 'Programs' which is just text.*" Three participants used both to give a more complete picture of the task, while one participant relied mostly on the programs because it gave more information. Most *Outcome-Diff: Questions* participants stated that they generally did not rely on seeing programs because the interface was easier to use. A handful relied on both the programs and the interface in order to get a more complete picture, while two relied mostly on the programs.

Most *Property-Diff* participants relied on the interface alone or with the programs. They found the interface easier to grasp while the programs had unnecessary information. Some mentioned that this preference varied between tasks.

7.4 Perception of Interface Usability

SUS scores did not differ significantly across interfaces. The mean score per interface ranged from 53.8 to 69.9, while the median ranged from 54.2 to 73.3. An SUS score of 68 is often considered average. Some interfaces' mean and median scores were at or below 68, though the difficulty of our tasks could have biased participants' perceptions negatively. We designed the tasks to highlight both benefits and drawbacks of each interface, so all participants had to complete tasks for which their interface was unhelpful. *Outcome-Diff: Flowcharts* participants were also significantly more likely to complete some of the complicated tasks correctly than control participants (Section 7.2), despite the low SUS rating of this interface.

7.5 Study Experiences

In total, participant perception of how demanding the tasks were differed significantly ($\chi^2(4) = 12.319$, $p = 0.015$), with *Outcome-Diff: Questions* participants finding tasks significantly less demanding compared to *Rules* participants ($p = 0.014$). This perception differed significantly between interfaces for Task 3 (Redundant Programs) ($\chi^2(4) = 18.987$, $p = 0.005$), where *Text-Diff* ($p = 0.033$), *Outcome-Diff: Flowcharts* ($p = 0.001$), and *Outcome-Diff: Questions* ($p = 0.001$) participants all found the tasks less demanding than *Rules*. For Task 5 (27 Variants) ($\chi^2(4) = 18.375$, $p = 0.005$), *Outcome-Diff: Questions* participants found the task less demanding than *Outcome-Diff: Flowcharts* participants ($p = 0.002$). For Task 6 (Abstraction) ($\chi^2(4) = 18.834$, $p = 0.005$), *Outcome-Diff: Questions* ($p = 0.015$) and *Property-Diff* ($p = 0.003$) participants found the task less demanding than *Rules*. Figure 12 shows participant ratings of how demanding tasks were.

Although participants were more likely to answer tasks correctly using the novel interfaces than the control interfaces, participants' confidence in their answers did not vary across tasks, either for any individual task or in aggregate across tasks. While they were not significantly more likely to answer tasks correctly, male participants were significantly more confident in their answers than female participants ($p = 0.044$) in our regression model.

Similarly, user perception of interface helpfulness did not differ significantly overall or for any task. Emphasizing the potential benefit of user interfaces that support TAP, participants without a technical background found the interfaces more helpful than those with such a background ($p = 0.017$) in our regression model. Furthermore, participants who did not own IoT devices found the interfaces more helpful than those who did ($p < 0.001$).

8 DISCUSSION AND CONCLUSION

To help users reason about differences in TAP programs, we designed a set of complementary interfaces that contrast TAP variants at different levels of semantic abstraction. We conducted an online user study to compare semantic-diff interfaces (*Outcome-Diff: Flowcharts*, *Outcome-Diff: Questions*, and *Property-Diff*) to two traditional interfaces (*Rules* and *Text-Diff*) in helping users complete TAP tasks with different characteristics. In our online user study,

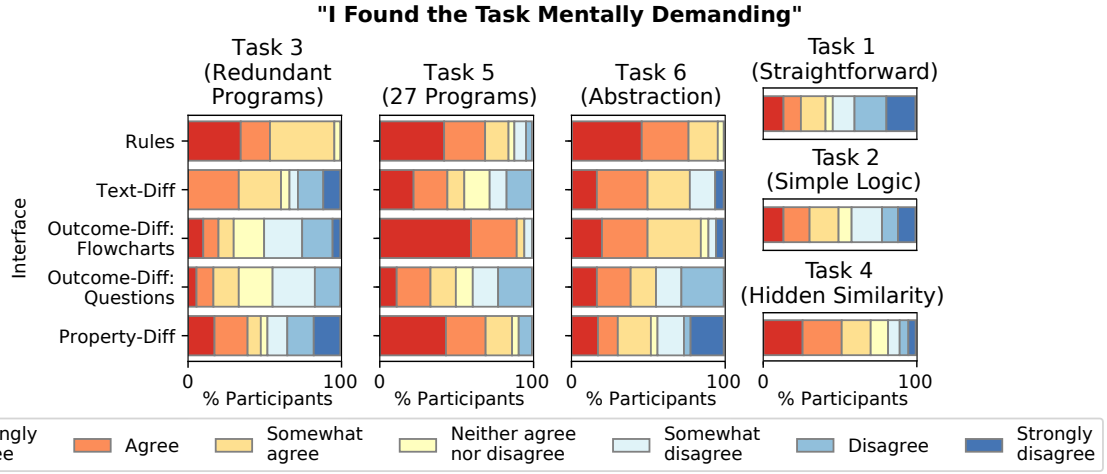


Figure 12: The distribution of participants ratings, by interface, for how demanding they considered each task. For Tasks 1, 2, and 4, ratings did not vary significantly by interface, so we present only the aggregate distribution for all participants.

we found that participants could correctly reason about differences between variants of short, simple programs by examining only the rules themselves. However, when comparing variants of long, complex programs, participants using semantic-diff interfaces significantly outperformed those using *Rules* or *Text-Diff*. *Outcome-Diff: Flowcharts* participants performed better when the task required identifying a manageable number of situation-specific differences in complex programs, while *Property-Diff* participants performed better when reasoning about more abstract differences. Participants using the low-level interfaces, *Outcome-Diff: Flowcharts* and *Outcome-Diff: Questions*, were able to identify program differences correctly across a wider variety of tasks than other users. *Outcome-Diff: Questions* participants often found tasks less demanding than others and significantly outperformed *Outcome-Diff: Flowcharts* participants when comparing many variants.

Our work advances intuitive methods of surfacing smart-system behaviors to users. We show that TAP interfaces should visualize information at multiple levels of granularity. By having automated reasoning using formal methods underpin user interfaces for end-user programming, the community can help TAP users better match a system's behaviors to their intent. **To facilitate future work, our open-source code for the interfaces, survey instrument, and regression tables are available online [53].**

8.1 Deployment Recommendations

Our interfaces have complementary strengths and weaknesses based on the characteristics of the TAP variants being compared. We believe that real-world deployments should take these trade-offs into account, showing the user a diff interface appropriate for particular situation and set of variants they are comparing. Fully understanding how to approximate user intent and identify the relevant characteristics of the TAP variants in order to automatically select a contextually appropriate diff interface requires future research, as does better understanding the potential usability confounds of showing a single user different interfaces based on the situation. Nonetheless, our results provide initial suggestions for contextually appropriate interfaces. When a user is comparing variants of short programs (each with a few rules and each rule with a few conditions), they should simply view the rules themselves. As programs becomes longer and more complex during the modification process, the system should present semantic-diff interfaces to help users understand the effects of their modifications. Because *Outcome-Diff: Questions* helped participants in our study reason about a wider variety of tasks and made the tasks less demanding, it could perhaps be presented by default. If the user is choosing from many variants, an interactive form-based interface like *Outcome-Diff: Questions* will help them eliminate undesirable choices faster.

Rather than trying to automate the selection of an appropriate interface, the system could instead ask the user whether they want to see differences in situation outcomes or high-level trends (properties). To account for *Property-Diff* participants' reliance on viewing the rules via the program button in our study, *Property-Diff* interfaces should also provide prominent access to the rules themselves, perhaps even displaying them by default. More broadly, we found that participants sometimes struggled to understand how the properties and situations the semantic interfaces displayed related to the trigger-action rules. We recommend that real-world deployments further clarify this connection, perhaps in an expanded tutorial introduction before walking the user through concrete interfaces.

Our interfaces could also perhaps be useful during program creation. To minimize repeated computation and facilitate just-in-time diffs after each change, future work should incrementally build transition systems rather than producing a new one from scratch each time. Future work can also consider applying these interfaces outside of trigger-action programming, such as to outcome-based program synthesis or constraint-based programming.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants CCF-1837120, CCF-1836948, and OAC-1835890. It was also supported by a gift from the CERES Research Center and the Marian and Stuart Rice Research Award.

REFERENCES

[1] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013. Automated Grading of DFA Constructions. In *Proc. IJCAI*.

[2] Koosha Araghi. 2014. *Google Docs Has Full 'Track Changes' Word Integration*. Retrieved July 8, 2019 from https://www.upcurvecloud.com/blog/google-docs-has-full-track-changes-word-integration/

[3] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.

[4] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. 2019. How Users Interpret Bugs in Trigger-Action Programming. In *Proc. CHI*.

[5] A.J. Bernheim Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. 2011. Home Automation in the Wild: Challenges and Opportunities. In *Proc. CHI*.

[6] Nico Castelli, Corinna Ogonowski, Timo Jakobi, Martin Stein, Gunnar Stevens, and Volker Wulf. 2017. What Happened in My Home? An End-User Development Approach for Smart Home Data Visualization. In *Proc. CHI*.

[7] Ryan Chard, Kyle Chard, Jason Alt, Dilworth Y. Parkinson, Steve Tuecke, and Ian Foster. 2017. Ripple: Home Automation for Research Data Management. In *Proc. ICDCSW*.

[8] Ryan Chard, Rafael Vescovi, Ming Du, Hanyu Li, Kyle Chard, Steve Tuecke, Narayanan Kasthuri, and Ian Foster. 2018. High-Throughput Neuroanatomy and Trigger-Action Programming: A Case Study in Research Automation. In *Proc. AI-Science*.

[9] Sven Coppers, Davy Vanacken, and Kris Luyten. 2020. FORTNIoT: Intelligible Predictions to Improve User Understanding of Smart Home Behavior. *PACM IMWUT* 4, 4 (2020).

[10] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. Empowering End Users in Debugging Trigger-Action Rules. In *Proc. CHI*.

[11] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. My IoT Puzzle: Debugging IF-THEN Rules Through the Jigsaw Metaphor. In *Proc. IS-EUD*.

[12] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How Can Automatic Feedback Help Students Construct Automata? *TOCHI* (2015).

[13] Ben Francis. 2019. *Introducing Mozilla WebThings*. Retrieved July 8, 2019 from https://hacks.mozilla.org/2019/04/introducing-mozilla-webthings/

[14] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. 2017. Personalization of Context-Dependent Applications Through Trigger-Action Rules. *TOCHI* (2017).

[15] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *Proc. TOCHI* (2015).

[16] Reid Holmes and Gail C. Murphy. 2005. Using Structural Context to Recommend Source Code Examples. In *Proc. ICSE*.

[17] Justin Huang and Maya Cakmak. 2015. Supporting Mental Model Accuracy in Trigger-action Programming. In *Proc. UbiComp*.

[18] IEEE and The Open Group. 2018. *diff*. Retrieved Jan 5, 2019 from https://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html

[19] IFTTT. 2019. *IFTTT helps your apps and devices work together*. Retrieved July 6, 2019 from https://ifttt.com/

[20] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *Proc. ICSE*.

[21] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-user Software Engineering. *Comput. Surveys* (2011).

[22] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proc. CHI*.

[23] Sandeep Kaur Kuttal, Anita Sarma, Gregg Rothermel, and Zhendong Wang. 2018. What happened to my application? Helping end users comprehend evolution through variation management. *IST* (2018).

[24] Abner Li. 2019. *Google Docs rolling out dedicated 'Compare Documents' tool*. Retrieved July 8, 2019 from https://9to5google.com/2019/06/11/google-docs-compare-documents/

[25] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F. Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically Debugging IoT Control System Correctness for Building Automation. In *Proc. BuildSys*.

[26] Marco Manca, Fabio, Paternò, Carmen Santoro, and Luca Corcella. 2019. Supporting end-user debugging of trigger-action rules for IoT applications. *IJHCS* (2019).

[27] Edward J. McCluskey. 1986. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Prentice-Hall, Inc., USA.

[28] Sarah Mennicken, David Kim, and Elaine May Huang. 2016. Integrating the Smart Home into the Digital Calendar. In *Proc. CHI*.

[29] Aaron Meurer et al. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* (2017).

[30] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An Empirical Characterization of IFTTT: Ecosystem, Usage, and Performance. In *Proc. IMC*.

[31] Mozilla. 2018. *Announcing "Project Things" - An open framework for connecting your devices to the web*. Retrieved July 8, 2019 from https://blog.mozilla.org/blog/2018/02/06/announcing-project-things-open-framework-connecting-devices-web/

[32] MSFTMan. 2019. *Create a flow in Microsoft Flow*. Retrieved July 8, 2019 from https://docs.microsoft.com/en-us/flow/get-started-logic-flow

[33] Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press.

[34] Christine M. Neuwirth, Ravinder Chandhok, David S. Kaufer, Paul Erion, James Morris, and Dale Miller. 1992. Flexible Diff-Ing in a Collaborative Writing System. In *Proc. CSCW*.

[35] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *TOCE* (2019).

[36] Mark Otto. 2014. *Introducing split diffs*. Retrieved July 6, 2019 from https://github.blog/2014-09-03-introducing-split-diffs/

[37] Stefan Palan and Christian Schitter. 2018. Prolific.ac - A subject pool for online experiments. *JBEF* (2018).

[38] Mitali Palekar, Earlence Fernandez, and Franziska Roesner. 2019. Analysis of the Susceptibility of Smart Home Programming Interfaces to End User Error. *Proc. SafeThings* (2019).

[39] Jamie Peabody. 2019. *Diff text documents online with Mergely, an editor and HTML5 javascript library*. Retrieved July 6, 2019 from http://www.mergely.com/

[40] Eyal Peer, Laura Brandimarte, Sonam Samat, and Alessandro Acquisti. 2017. Beyond the Turk: Alternative platforms for crowdsourcing behavioral research. *JESP* (2017).

[41] Andrea Piscitello, Alessandro A. Nacci, Vincenzo Rana, Marco D. Santambrogio, and Donatella Sciuto. 2016. Ruleset Minimization in Multi-tenant Smart Buildings. In *Proc. CSE and EUC and DCABES*.

[42] Justin Pot. 2019. *What Is Version History and How to Use It in Google Docs?* Retrieved July 8, 2019 from https://zapier.com/apps/google-docs/tutorials/google-docs-revision-history

[43] Amir Rahmati, Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2017. IFTTT vs. Zapier: A Comparative Study of Trigger-Action Programming Frameworks. *CoRR* (2017).

[44] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proc. WWW*.

[45] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *Proc. VL/HCC*.

[46] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical Trigger-action Programming in the Smart Home. In *Proc. CHI*.

[47] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *Proc. CHI*.

[48] Svetlana Yarosh and Pamela Zave. 2017. Locked or Not?: Mental Models of IoT Feature Interaction. In *Proc. CHI*.

[49] Zapier. 2019. *How Zapier Works*. Retrieved July 6, 2019 from https://zapier.com/help/how-zapier-works/

[50] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: Synthesizing and Repairing Trigger-action Programs Using LTL Properties. In *Proc. ICSE*.

[51] Lefan Zhang, Weijia He, Olivia Morkved, Valerie Zhao, Michael L. Littman, Shan Lu, and Blase Ur. 2020. Trace2TAP: Synthesizing Trigger-Action Programs from Traces of Behavior. *PACM IMWUT* 4, 3 (2020).

[52] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proc. ESEC/FSE*.

[53] Valerie Zhao, Lefan Zhang, Bo Wang, Michael L. Littman, Shan Lu, and Blase Ur. 2021. Supplementary Materials for Understanding Trigger-Action Programs Through Novel Visualizations of Program Differences. https://www.blaseur.com/papers/chi21-programdiff-appendix.pdf