Visualizing Differences to Improve End-User Understanding of Trigger-Action Programs

Valerie Zhao
University of Chicago
vzhao@uchicago.edu

Shan Lu
University of Chicago
shanlu@uchicago.edu

Lefan Zhang
University of Chicago
lefanz@uchicago.edu

Blase Ur
University of Chicago
blase@uchicago.edu

Bo Wang
University of Science and Technology of China / University of Chicago
wbhalo@mail.ustc.edu.cn

Abstract

Trigger-action programming lets end-users automate and connect IoT devices and online services through if-this-then-that rules. Early research demonstrated this paradigm's usability, but more recent work has highlighted complexities that arise in realistic scenarios. As users manually modify or debug their programs, or as they use recently proposed automated tools to the same end, they may struggle to understand how modifying a trigger-action program changes its ultimate behavior. To aid in this understanding, we prototype user interfaces that visualize differences between trigger-action programs in syntax, behavior, and properties.

Author Keywords

trigger-action programming; smart homes; IoT; automation

CCS Concepts

•**Human-centered computing** $\rightarrow$ **User interface programming;**

Introduction

Advances in end-user programming have enabled people, regardless of their technical background, to express logic to computer systems [24, 31]. For example, trigger-action programming (TAP) is an end-user programming paradigm centered on if-this-then-that rules that has become widespread in recent years [38]. In TAP, a *program* consists of one or more *rules* in the form of "*If* trigger *While* conditions *Then* action." Thus, in each rule the user specifies a triggering event, all conditions (if any) that must be true when that event occurs, and the action that should be carried out. For example, a user who wishes to turn on the lights when they enter the living room at night may write, "*If* I enter the living room *While* it is nighttime *Then* turn on the living room lights." TAP rules empower end-users to automate smart homes [29, 39], online social media [37, 43], and scientific research [8, 9]. TAP is supported by services such as Microsoft Flow [30], Zapier [43], Mozilla WebThings [14], and IFTTT [21]—with the latter boasting 11 million users and 54 million applets as of July 2019 [22].

TAP is ubiquitous and intuitive in straightforward situations [15, 38], yet its event-driven nature can cause challenges and result in buggy programs in common scenarios [4, 18, 42]. Debugging, modifying, or even understanding TAP programs can be complicated for many reasons. First, satisfying multiple goals at once requires numerous TAP rules [17], resulting in crowded user interfaces and complex interactions between rules. Furthermore, when TAP is deployed in smart homes—typically shared spaces—multiple users may contribute rules. They may not understand how each other's rules impact their home or why certain events have occurred [28]. Users might be reluctant to adjust a program due to fear of messing up the system [16]. Finally, a number of recent efforts have sought to use techniques based on formal methods [7, 26, 40, 44] to analyze and, in some cases, modify TAP programs automatically.

Collectively, the resulting errors can lead to discomfort, wasted resources, monetary and time costs, friction between users, and security risks. Users currently have little support for identifying and fixing such errors in the wild. In smart homes, users typically must rely on trial-and-error to debug programs [27, 41], and some bugs may only surface occasionally. To eliminate bugs or introduce new features, end-users adapt existing code [5, 24] and modify software in an iterative process [6, 27, 41].

Program 1:

- If the AC turns on while the window is open then turn off the AC.
- If I leave the bathroom while the smart faucet is on then turn off the smart faucet.

Program 2:

- If the AC turns on while the window is open then **close the window**.
- If I leave the bathroom while the smart faucet is on **and it is nighttime** then turn off the smart faucet.

Figure 1: Variants of a TAP program. Text differences are bolded. Neither program contains a rule that affects the other rule in the same program.

We observe that TAP rules in the above scenarios—those created by tools, by other members of a household, or through tweaks by the users themselves—result in changes to existing programs. Our key insight is that users must understand differences between the pre- and post-modified versions of a program, in both implementation details (the rules) and in their effect on the system. Minimal research on TAP has focused on these differences between TAP program variants despite its importance in the end-user experience. Automated tools and actions by others impair a user's ability to understand the system's behavior. Making system behaviors more transparent and intelligible would aid trust [3] and comfort with the system [23]. It could also better aid in the reuse of TAP programs [5].

In this extended abstract, we present a series of prototype interfaces that visualize various nuances of the differences between variants of a TAP program. Our interfaces compare and contrast TAP variants to help end-users understand their differences. Regardless of experience, users can utilize these interfaces to understand changes made to a program by themselves or others. Our prototype interfaces cover three levels of granularity: low-level changes in the programs' text, slightly higher-level differences in actions taken by the system in specific scenarios, and high-level properties. These interfaces complement each other to provide a holistic view of the variants and the resulting system. We developed the concepts for these interfaces based on scenarios from the TAP literature [4, 11, 12, 18, 42] where understanding the differences between variants of a program could aid in debugging. We further based the

1. If I wake up while my roommate is awake then turn the lights on.
2. If my roommate wakes up while the lights are off and I am awake then turn the lights on.
3. If the lights turn off while I am awake then turn the lights on.
4. If I wake up while my roommate is asleep and the lights are off then turn the lights on.
5. If the lights turn off while my roommate is awake and I am awake then turn the lights on.
6. If the lights turn off while I am asleep and my roommate is awake then turn the lights on.
7. If my roommate wakes up then turn the lights on.

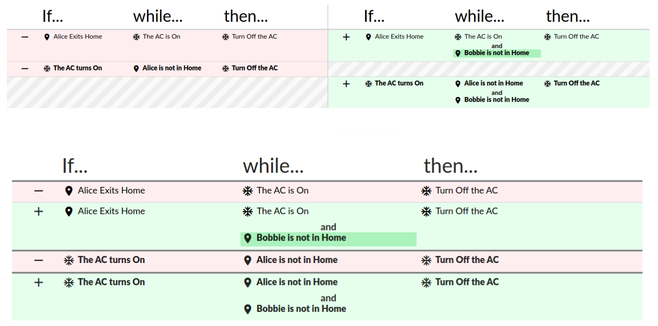
Figure 2: A redundant TAP program that ensures the lights are always on when "I am asleep," "my roommate is asleep," or both.

visual designs on "diff" utilities for code [20, 32] and collaborative word-processing tools [1, 35]. We hypothesize that these interfaces will help users identify differences in the behaviors of a single smart device, interactions between devices, and a full TAP system's long-term guarantees.

Interface Design and Implementation

In this section, we describe our interfaces for comparing and contrasting TAP program variants. First, we present a pair of interfaces that visualize text differences to compare low-level implementation details (with respect to the paradigm syntax itself). They may help the user comprehend simple behaviors of individual smart devices. Next, we present a pair of interfaces for contrasting system behavior, specifically the actions triggered under identical scenarios, to help users reason about more complex interactions between devices. Namely, we present a flowchart-based interface that visualizes behavior differences between two variants, and a form-based interface that enables users to directly define desired behaviors among multiple variants. Finally, we present an interface that highlights differences in properties held by each variant, which can help the user understand long-term guarantees of the resulting system.

Our goals were informed by observations of end-user practices [6, 27, 41] and common bugs [4, 18, 42] in TAP. When applicable, we reviewed existing ideas in software engineering—namely visualizations for code and text differences [32, 34, 35] and execution traces [13, 33]—and applied them to our designs. We then iteratively improved our designs through informal pilot studies in person and on crowdsourced online platforms. For interfaces that highlight differences beyond text, we model devices and programs as transition systems [2, 44], then conduct graph analysis to deduce differences.



	If...	while...	then...		If...	while...	then...
—	Alice Exits Home	The AC is On	Turn Off the AC	+	Alice Exits Home	The AC is On and **Bobbie is not in Home**	Turn Off the AC
—	**The AC turns On**	**Alice is not in Home**	**Turn Off the AC**				
				+	**The AC turns On**	**Alice is not in Home** and **Bobbie is not in Home**	**Turn Off the AC**

	If...	while...	then...
—	Alice Exits Home	The AC is On	Turn Off the AC
+	Alice Exits Home	The AC is On and **Bobbie is not in Home**	Turn Off the AC
—	**The AC turns On**	**Alice is not in Home**	**Turn Off the AC**
+	**The AC turns On**	**Alice is not in Home** and **Bobbie is not in Home**	**Turn Off the AC**

Figure 3: Side-by-side (top) and integrated (bottom) text differences modeled after Github [32], which uses red to represent removed or modified rule clauses on the first program, and green for added or modified rule clauses on the second program.

Text Difference

Sometimes users may simply wish to compare code based on text. Figure 1 illustrates an example where, not only are the programs almost identical, but the rules within a program do not interact with each other. In this case, because the difference between the rules are straightforward—neither program contains rules that interact with each other—an interface that simply highlights text differences between programs may be most helpful. Therefore, we designed two interfaces (Figure 3) modeled after Github's diff views [32] to compare TAP programs based on syntax.

For every pair of programs, we treat the second program as the result of modifying the first regardless of the actual provenance. This is similar to the idea of edit distance [25], which is well-established in analyzing how a text string has changed. However, based on participant feedback during the pilot, nontechnical users were unaccustomed to this way of thinking. Thus, we minimized mentioning these ideas in the actual interfaces. We first determine the rules shared

1. If I wake up while the lights are off then turn the lights on. (1, 4)
2. If my roommate wakes up while the lights are off then turn the lights on. (7)
3. If the lights turn off while I am awake then turn the lights on. (3)
4. If the lights turn off while my roommate is awake then turn the lights on. (5, 6)

Figure 4: A succinct program that, like the redundant program in Figure 2, also ensures the lights are always on when “I am asleep,” “my roommate is asleep,” or both. In this case, each rule is equivalent to one or a combination of multiple rules in the redundant program. Numbers in parentheses map to the redundant program rules in Figure 2.

by programs using simple set operations. If one program is a complete subset of another, we highlight the unshared rules as “removed” (if belonging to the first program) or “added” (if in the second). However, if a pair of similar yet distinct rules exist, such as two rules that share the same trigger but differ in the action, we determine whether we can categorize them as “modified.” Our criteria for “modified” rules is that the two rules differ in only one clause: only the trigger, the list of conditions, or the action is different.

Behavior Difference: Flowchart-Based Interface
Under identical scenarios, different TAP programs may trigger different actions. For example, if it becomes nighttime while someone is home, one program might turn on the lights while another might not. A user may wish to choose between these two behaviors. However, in complex, realistic systems where there can be many trigger-action rules and complex interactions between them, tracking variables like “is it nighttime” or “whether someone is at home” can be difficult. In particular, two programs may result in the same system states in all scenarios, despite redundant rules or different actions (e.g. the programs in Figures 2 and 4). To help users with this, we designed a flowchart-based interface (Figure 5) to describe the scenarios in which the system will take different actions.

For each pair of programs, we walk the user through scenarios in which the programs would take different actions. From left to right, we first show a state of the system before any rules are triggered. This system state is a set of the states of the individual devices at that point in time, such as whether it is nighttime and whether someone is at home. We then show that a triggering event occurs, such as the day becoming nighttime. This event can come from the environment (“it starts raining”) or people (“someone opens the window”). We also show the resulting state of the system. Finally, the flowchart diverges. For each program, we show the action it takes and the consequent system state, while highlighting all device states that differ from those of the other program. While the interface will only show scenarios in which the two programs take different actions, it is possible for the actions to result in the same final system state. In this case, the flowchart will not diverge.

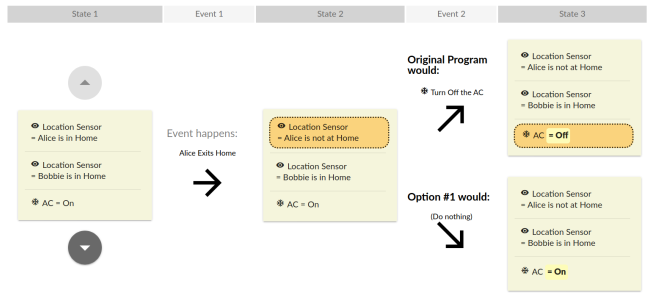

State 1
Event 1
State 2
Event 2
State 3
Location Sensor = Alice is in Home
Location Sensor = Bobbie is in Home
AC = On
Event happens:
Alice Exits Home
Location Sensor = Alice is not at Home
Location Sensor = Bobbie is in Home
AC = On
Original Program would:
Turn Off the AC
Option #1 would:
(Do nothing)
Location Sensor = Alice is not at Home
Location Sensor = Bobbie is in Home
AC = Off
Location Sensor = Alice is not at Home
Location Sensor = Bobbie is in Home
AC = On

Figure 5: A flowchart-based difference interface. State 1, State 2, and Event 1 describe the scenario setup that triggers different actions based on the two programs. Devices whose states have been affected by the previous event are highlighted in orange with a dotted line around them. The device states at the end that differ based on the two programs are highlighted in yellow.

Behavior Difference: Form-Based Interface
Users may sometimes wish to compare more than two programs. Suppose that a user specifies the following goal to a TAP program synthesizer: the living room lights should never be on while it is daytime and the window curtains are open. That is, at most two of the following can be true simultaneously: living room lights on, daytime, and window curtains open. Without any other constraints, the synthesizer should produce eight possible programs by permutating over whether the system turn off the lights or close the window curtains for each rule, e.g. Program 1 in Figure 6.

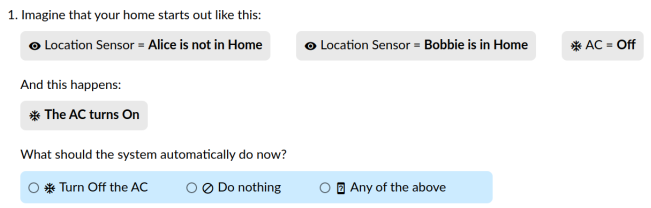
Program 1:

1. If the window curtains open while the lights are on and it is daytime then turn the lights off.
2. If the lights turn on while it is daytime and the window curtains are open then turn the lights off.
3. If it becomes daytime while the lights are on and the window curtains are open then turn the lights off.

Program 2:

1. If the window curtains open while the lights are on and it is daytime then turn the lights off.
2. If the lights turn on while it is daytime and the window curtains are open then turn the lights off.

Figure 6: A pair of programs for which the properties differ. Program 1 ensures the lights are never on while it is daytime and the window curtains are open. Program 2 does not ensure this because it is missing the last rule.



1. Imagine that your home starts out like this:

Location Sensor = **Alice is not in Home** | Location Sensor = **Bobbie is in Home** | AC = **Off**

And this happens:

The AC turns On

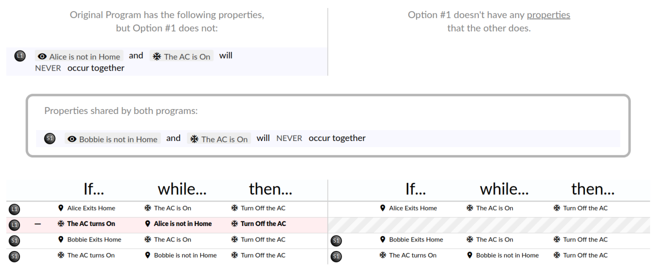
What should the system automatically do now?

○ Turn Off the AC ○ Do nothing ○ Any of the above

Figure 7: A sample question on the form-based interface. The options "Turn off the AC" and "Do nothing" correspond to actions that the two programs would take. Choosing "Any of the above" will discard this particular scenario from further consideration.

In cases where users want to compare more than two programs, such as the example above, it is tiresome and tedious for the user to compare every pair of the programs. Users may also have goals that they failed to articulate to automated tools to further constrain their options [24]. We address this issue by utilizing a multiple-choice form (Figure 7) that compares the programs and elicits user-desired, scenario-specific goals. Each question on the form describes a scenario in which at least two programs under consideration would take different actions. For each question, the user can choose between the actions (one of which could be "do nothing"), or indicate that they do not care about the action to take. The user answers the questions until there are no more differences left. The interface calculates the program(s) that satisfy the most answers from the user, and present those program(s) to the user.

Currently, the interface does not attempt to minimize the number of questions asked, which can be done by showing one question at a time and then eliminating programs that do not behave according to the answer for the previous question. However, we chose not to do so in case users



Original Program has the following properties, but Option #1 does not:

L1 Alice is not in Home and The AC is On will NEVER occur together

Option #1 doesn't have any properties that the other does.

Properties shared by both programs:

S1 Bobbie is not in Home and The AC is On will NEVER occur together

		If...	while...	then...		If...	while...	then...
L1		Alice Exits Home	The AC is On	Turn Off the AC		Alice Exits Home	The AC is On	Turn Off the AC
L1	—	**The AC turns On**	**Alice is not in Home**	**Turn Off the AC**				
S1		Bobbie Exits Home	The AC is On	Turn Off the AC	S1	Bobbie Exits Home	The AC is On	Turn Off the AC
S1		The AC turns On	Bobbie is not in Home	Turn Off the AC	S1	The AC turns On	Bobbie is not in Home	Turn Off the AC

Figure 8: A property-based difference interface. The top half lists shared and different properties between the two programs. The bottom half shows the side-by-side text differences. If a rule has a black circle label to its left, it is a rule that contributed to the property marked with the same label.

implicitly prefer certain scenarios over others. We leave scenario prioritization to future work.

Property Difference

Finally, sometimes users may want to know about long-term patterns or guarantees in the system. For example, a user who has to choose between the two programs in Figure 6 may wish that the lights are never on while it is daytime and the window curtains are open. However, current TAP interfaces do not empower users to specify such long-term goals. Furthermore, manually going through the programs and the possible scenarios to determine which program satisfies some goal can be difficult and error-prone. Zhang et al. [44] found that many smart home user requirements can be expressed in the form of a safety property, such as, "The lights should never be on while it is daytime and the window curtains are open." With these factors in mind, we developed an interface to compare and contrast the safety properties held by a pair of programs (Figure 8). The safety property templates we support come from Zhang et al. [44].

This interface builds off of the side-by-side text difference interface (see top of Figure 3). We chose to do so based on participant feedback from our pilots, which suggested that users preferred seeing the rules rather than only the properties. In isolation, properties abstract too much information away from what the system actually does. We further address this concern by indicating which rules contribute to which properties, demonstrating to users that the properties shown are correct. We deduce the shared and unshared properties of each pair of programs by conducting reachability analysis on the underlying transition systems.

Future Work

We will validate the utility of these interfaces through an online user study. We will give users tasks that mimic user experiences with smart home automation, asking them to debug, extend, or shorten programs to meet given requirements. Some scenarios will be based on common bugs identified in the literature [4, 18, 42]. The tasks will range in complexity and nuance of differences, and a standalone view of each TAP program will serve as the control. To determine whether the interfaces are helpful and for which scenarios they are most helpful, we will analyze the completion time and accuracy for participants using each interface.

Related Work

Widely used interfaces visualize differences in text code (e.g., on Github [32]). Other work tries to visualize automated system behaviors, as described below. To the best of our knowledge, however, little work exists in the intersection of these two fields. Such an intersection attempts to visualize end-user program differences *beyond* text to aid user comprehension of TAP programs. The simplicity of TAP programs relative to general programs makes it feasible to identify and visualize their differences *beyond* text.

Existing tools for mitigating TAP issues mostly leverage traditional software engineering techniques. They also mostly focus on translating end-users' automation goals into TAP programs and identifying and fixing bugs in TAP programs. To understand and translate what users want to achieve with the TAP programs, some prior work has leveraged natural language processing [10, 36], crowdsourcing [19], and formal methods [44]. For debugging TAP programs, multiple tools [7, 17, 26, 37, 40, 44] leverage formal methods or information flow analysis to check a given program for vulnerabilities and adherence to user requirements.

As some users are still reluctant to use TAP out of a fear of breaking the system [16], methods of surfacing information about the system's internal state will be helpful. Some work has been done to help end-users understand TAP programs by displaying their effect on the smart home on a calendar [28], displaying rules as a jigsaw [12], and offering step-by-step explanations of automatically identifiable TAP bugs [11, 12]. Our work builds upon this literature to support user understanding by contrasting TAP program variants.

Conclusion

Smart home devices can be shared among users who automate them in ways that conflict. Rules in TAP programs can interact in complex ways. Recently proposed debugging tools try to automatically fix TAP programs, yet users may not trust these fixes. As a result, users often need to compare highly related variants of a TAP program, yet receive little support from current interfaces. We facilitate end-user understanding of TAP differences by introducing interfaces that help users compare and contrast TAP program variants. Our interfaces help users reason about syntax differences, differences in actions under identical scenarios, and property differences. We plan to improve and validate the usability of these interfaces through user studies.

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grants 1837120 and 1835890, as well as a gift from the CERES Center for Unstoppable Computing. We thank Olivia Morkved for her help conducting the pilot studies.

REFERENCES

[1] Koosha Araghi. 2014. Google Docs Has Full 'Track Changes' Word Integration. (12 2014). Retrieved July 8, 2019 from https://www.upcurvecloud.com/blog/google-docs-has-full-track-changes-word-integration/

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

[3] Victoria Bellotti and Keith Edwards. 2001. Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Hum.-Comput. Interact.* 16, 2 (Dec. 2001), 193–212. DOI: http://dx.doi.org/10.1207/S15327051HCI16234_05

[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 *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 552, 12 pages. DOI: http://dx.doi.org/10.1145/3290605.3300782

[5] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover. *IEEE Software* 26, 5 (Sept. 2009), 18–24. DOI: http://dx.doi.org/10.1109/MS.2009.147

[6] 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 *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. Association for Computing Machinery, New York, NY, USA, 2115–2124. DOI: http://dx.doi.org/10.1145/1978942.1979249

[7] Lei Bu, Wen Xiong, Chieh-Jan Mike Liang, Shi Han, Dongmei Zhang, Shan Lin, and Xuandong Li. 2018. Systematically Ensuring the Confidence of Real-Time Home Automation IoT Systems. *ACM Trans. Cyber-Phys. Syst.* 2, 3, Article Article 22 (June 2018), 23 pages. DOI: http://dx.doi.org/10.1145/3185501

[8] R. Chard, K. Chard, J. Alt, D. Y. Parkinson, S. Tuecke, and I. Foster. 2017. Ripple: Home Automation for Research Data Management. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 389–394. DOI: http://dx.doi.org/10.1109/ICDCSW.2017.30

[9] 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 *Proceedings of the 1st International Workshop on Autonomous Infrastructure for Science (AI-Science'18)*. ACM, New York, NY, USA, Article 1, 7 pages. DOI: http://dx.doi.org/10.1145/3217197.3217206

[10] Xinyun Chen, Chang Liu, Richard Shin, Dawn Song, and Mingcheng Chen. 2016. Latent Attention for If-Then Program Synthesis. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., USA, 4581–4589. http://dl.acm.org/citation.cfm?id=3157382.3157609

[11] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019a. Empowering End Users in Debugging Trigger-Action Rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 388, 13 pages. DOI: http://dx.doi.org/10.1145/3290605.3300618

[12] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019b. My IoT Puzzle: Debugging IF-THEN Rules Through the Jigsaw Metaphor. In *End-User Development*, Alessio Malizia, Stefano Valtolina, Anders Morch, Alan Serrano, and Andrew Stratton (Eds.). Springer International Publishing, Cham, 18–33.

[13] Stephan Diehl. 2007. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag, Berlin, Heidelberg.

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

[15] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. 2017. Personalization of Context-Dependent Applications Through Trigger-Action Rules. *ACM Trans. Comput.-Hum. Interact.* 24, 2, Article 14 (April 2017), 33 pages. DOI: http://dx.doi.org/10.1145/3057861

[16] Weijia He, Jesse Martinez, Roshni Padhi, Lefan Zhang, and Blase Ur. 2019. When Smart Devices Are Stupid: Negative Experiences Using Home Smart Devices. *Proceedings of the SafeThings Workshop* (Jan 2019). http://par.nsf.gov/biblio/10095907

[17] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. 2019. SAFECHAIN: Securing Trigger-Action Programming from Attack Chains (Extended Technical Report). *CoRR* abs/1903.03760 (2019). http://arxiv.org/abs/1903.03760

[18] Justin Huang and Maya Cakmak. 2015. Supporting Mental Model Accuracy in Trigger-action Programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 215–225. DOI:http://dx.doi.org/10.1145/2750858.2805830

[19] Ting-Hao K. Huang, Amos Azaria, Oscar J. Romero, and Jeffrey P. Bigham. 2019. InstructableCrowd: Creating IF-THEN Rules for Smartphones via Conversations with the Crowd. *Human Computation* (2019), 101–131.

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

[21] IFTTT. 2019a. IFTTT helps your apps and devices work together. (2019). Retrieved July 6, 2019 from https://ifttt.com/

[22] IFTTT. 2019b. IFTTT Platform. (2019). Retrieved July 6, 2019 from https://platform.ifttt.com/lp/learn_more

[23] Antonio Isalgue, Massimo Palme, Helena Roura, and Rafael Serra. 2006. The Importance of Users' Actions for the Sensation of Comfort in Buildings. In *Proceedings of the PLEA Conference*.

[24] Andrew 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. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. DOI: http://dx.doi.org/10.1145/1922649.1922658

[25] VI Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707.

[26] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F. Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically Debugging IoT Control System Correctness for Building Automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys '16)*. Association for Computing Machinery, New York, NY, USA, 133–142. DOI: http://dx.doi.org/10.1145/2993422.2993426

[27] Sarah Mennicken and Elaine M. Huang. 2012. Hacking the Natural Habitat: An In-the-Wild Study of Smart Homes, Their Development, and the People Who Live in Them. In *Proceedings of Pervasive Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 143–160.

[28] Sarah Mennicken, David Kim, and Elaine May Huang. 2016. Integrating the Smart Home into the Digital Calendar. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 5958–5969. DOI: http://dx.doi.org/10.1145/2858036.2858168

[29] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An Empirical Characterization of IFTTT: Ecosystem, Usage, and Performance. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. ACM, New York, NY, USA, 398–404. DOI: http://dx.doi.org/10.1145/3131365.3131369

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

[31] Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing* (1st ed.). MIT Press, Cambridge, MA, USA.

[32] Mark Otto. 2014. Introducing Split Diffs. (Sept 2014). https://github.blog/2014-09-03-introducing-split-diffs/

[33] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. 2001. Visualizing the Execution of Java Programs. In *Revised Lectures on Software Visualization, International Seminar*. Springer-Verlag, Berlin, Heidelberg, 151–162.

[34] Jamie Peabody. 2019. Diff Text Documents Online With Mergely. (2019). http://www.mergely.com/

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

[36] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 878–888. DOI:`http://dx.doi.org/10.3115/v1/P15-1085`

[37] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1501–1510. DOI: `http://dx.doi.org/10.1145/3038912.3052709`

[38] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical Trigger-action Programming in the Smart Home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 803–812. DOI: `http://dx.doi.org/10.1145/2556288.2557420`

[39] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 3227–3231. DOI: `http://dx.doi.org/10.1145/2858036.2858556`

[40] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, New York, NY, USA, 1439–1453. DOI: `http://dx.doi.org/10.1145/3319535.3345662`

[41] Jong-bum Woo and Youn-kyung Lim. 2015. User Experience in Do-it-yourself-style Smart Homes. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 779–790. DOI:`http://dx.doi.org/10.1145/2750858.2806063`

[42] Svetlana Yarosh and Pamela Zave. 2017. Locked or Not?: Mental Models of IoT Feature Interaction. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 2993–2997. DOI: `http://dx.doi.org/10.1145/3025453.3025617`

[43] Zapier. 2019. How Zapier Works. (2019). `https://zapier.com/help/how-zapier-works/`

[44] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: Synthesizing and Repairing Trigger-action Programs Using LTL Properties. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 281–291. DOI: `http://dx.doi.org/10.1109/ICSE.2019.00043`