Building an Interface for Controlling IoT Devices

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI USA
spr@cs.brown.edu

***Abstract*—As more and more devices become available and interconnected, there is a growing need for end users to control the devices in a programmatic manner. Our prior work explored end-user programming models for the Internet of Things (IoT). We started with the premise that current trigger-based approaches, while helpful, are not sufficient. What is needed was a more programmatic, continuous approach for handling more complex devices. Our prior work explored different strategies for such a continuous approach. In this paper we detail the lessons learned and outline our current directions aimed at creating a practical system.**

***Index Terms*—Internet of things; end-user programming; debugging; program understanding.**

I. Introduction

It is becoming common for everyday devices, from televisions to toasters, to be connected to the Internet and controllable from anywhere. Most such devices provide separate apps that let end-users control them from their phones. While separate apps can provide a high-quality, device-specific interface, requiring tens or hundreds of apps is obviously not ideal for the user. Several alternatives have arisen. Some, like IFTTT [1], are generic, providing a trigger-based programming approach that is suitable for controlling simple devices. Others, like SmartThings [2] or OpenHab [3], let manufacturers create limited customizable interfaces for their devices and also provide a trigger-based programming environment for the devices. Another approach is to use a general command application such as Alexa or Siri. Manufacturers can also provide bridges from their devices to these, letting end-users control their devices using speech or simple time-based commands.

These approaches are insufficient. Prior work has shown that trigger-based programming approaches are not natural ways of specifying interactions [4,5]. Approaches using machine learning have been tried [6] and have been successful for single devices with a limited set of inputs such as the Nest thermostat. As more devices need to be controlled, the feature-interaction problem arises, so that languages for controlling them need to handle potentially complex interactions [7]. More importantly, from an end-user's perspective, is that a large number of rules may be needed for a complex device.

Our obsession is a "smart sign". This is a display outside our office showing our current status such as "Available" or "In a Meeting, Back at 1" or "On the Phone", and is updated automatically based on a variety of sensors and other devices. We currently have over 20 different status displays and most transitions from one to another are possible. If we were to program the sign using only a trigger-based approach, we would need hundreds of rules. Another example is a smart skylight that can be open, closed, or partially open. The complexity here comes from the set of sensors that such a skylight would depend on and the fact that these interact. The sensors might include the outside and inside temperatures, whether it is raining or not, the current inside and outside air quality, and the outside wind speed and direction. A trigger based approach to this would need rules for all possible changes under all possible conditions.

From this, we concluded that one needs a control language that describes the target state given current conditions rather than one that describes state changes based on all possible condition changes. This implies a rule-based, continuous approach as opposed to a trigger-based one. For our sign, for example, this means we need 20-some rules rather than hundreds. Such a language, while easy to conceive of, is not necessarily easy to implement, understand, or program.

Our goal is to provide a language for programming IoT devices using general rules, both continuous and trigger based. (Trigger-based rules are still needed for trigger-based actions such as sending an email or setting off an alarm.) The requirements of such a language start with being easy to use for non-programmers. It must be easy to understand what is written. It must be easy to understand the consequences of each rule on each device and the interactions of the various rules. It must work as an app on the user's phone (or similar device), with possible web support. An architecture supporting this language must be designed to work at Internet-scales, handling thousands of users and tens of thousands of devices.

We are currently attempting to demonstrate that such an approach can be practical. The next section of the paper describes our prior efforts. The following sections describe the lessons learned and our current approach.

II. Our Prior Work

Our prior work explored different end-user programming models for a continuous rule-based language for controlling IoT devices, with an emphasis on controlling our smart sign [8]. This work included four different web-based user interfaces, each with different goals in mind, an engine to interpret the rules, simple implementations of sensors (motion detector, off-the-hook detector, in the office detector using Bluetooth, working at home detector, on-zoom detector, Google calendar interface, web-based weather interface), and

a web-page implementation of our sign (the actual sign is a tablet running a kiosk app displaying that page). We also implemented experimental implementations of the engine with SmartThings and OpenHab.

The underlying engine supported priority-based rules that were run when any condition changed. It provided polling facilities for the devices that needed to be polled. It also provided a notion of hypothetical worlds where conditions could be set arbitrarily in order to explore possible rule conflicts and answer what-if questions. The engine only supported a single user and was designed to be run on that user's local machine. It also provided a RESTful web interface for the front ends.

The four interfaces explored different end-user programming goals. The first, a programmer's interface, provided a list of all the rules. It let the end-user change the rule priorities, edit rules, delete rules, and add new rules. The second, a new-rule interface, provided the front end for defining a new rule but also provided information as to what existing rules might conflict with the new rule as the user defined it. The third interface was a learning or demonstration based [6] interface based loosely on the Nest thermostat [9]. It let the user define a situation based on the available sensors and time, defaulting to current values, and then define the result. The system would then build a decision tree using this and all prior rules and would construct an appropriate rule. The final interface was a modular interface similar to the programmer's interface but where the end-user could restrict the set of displayed rules by what devices they affect, what devices they are affected by, or what conditions they are affected by. This was designed for debugging and attempted to provide a simpler interface in cases where there were a large number of rules.

While we are still using this system (the sign is currently available as https://www.cs.brown.edu/people/spr/status.html), we have not found it easy to use. The limitation to running on the user's machine limited its use to a single user and a single sign and required the user have a machine running 24/7. The different user interfaces, while each had distinct advantages, were each problematic in their own way. The programmer's interface was too complex, displaying too many rules, and it was easy to make mistakes. The new-rule interface was not always accurate and thus was not particularly helpful since it could not be trusted. The learning interface had problems because it had difficulty learning time intervals from events with specific times and very noisy data, the hypothetical worlds did not reflect history, and there were privacy concerns with keeping event data. The modular interface provided too much flexibility for creating rules and was confusing when there were rules that affected or were affected by multiple devices. Moreover, having four different user interfaces was not ideal.

To move forward, we needed to develop a practical, multi-user system with a single user interface that combined the good aspects of the existing system while avoiding its pitfalls. To this end, we first noted a number of lessons learned and are using these to design and build the follow-on system that will attempt to meet our requirements.

III. Lessons Learned

We have several years of experience with the current system and experimental user interfaces. Based on this we have discerned a number of precepts that need to be considered in developing a more practical approach. These apply to the underlying rule-based approach, the engine that interprets the rules, the user interfaces, and devices such as our smart sign.

A. Rule Lessons.

Rules are the basis for a continuous control-based language. They need to be defined in such a way that they can be easily understood and are amenable to simple, easy-to-use interfaces.

We first determined that rules should only handle AND conditions. AND is much more common than OR and combinations of AND and OR are difficult to understand, even for programmers. OR conditions can be emulated by defining new conditions or by multiple rules.

Next we understood the need for virtual conditions such as a latch (condition true once it is triggered), a duration (condition true for k minutes after triggering), or debouncing (condition has to be stable for k seconds to be considered). Our existing system implemented these both as virtual devices and as complex conditions. We found conditions easier to define and use. A practical approach must let the user define such conditions.

Considering all rules at once is confusing to the user even if there are only 20 some rules. The confusion comes in both understanding the priorities and how they interact, and in understanding what the rules do. The modular interface was useful here, but only when applied to a single device and then only if rules can only affect a single device. Since rules that affect multiple devices seem uncommon and can be easily mimicked as separate rules, we plan to restrict rules to a single device and the new interface will automatically provide modularity based on the device.

Priorities are confusing when there are a relatively large number of rules. Our experience shows that rules can be classified into meta-priority levels and fine priorities within these designations. Our new approach for devices with more than 5 rules is to predefine a set of 3-5 priority levels and let the user place rules within a level. They can still organize the rules within a level, but this should be a much smaller set and more easily understood.

Our current interface has the user provide names for rules for later display. While these names are helpful and necessary, their meaning is forgotten after years. In the existing system, users had to go to the editing interface to completely understand a rule. Our new system will augment names with a more detailed description of what the rule does.

Finally, we determined that the system needs to handle both triggers and continuous rules. To do so means that we need both trigger and continuous conditions, and need to specify, for each condition, whether it is a trigger or continuous. Actions can be triggered, continuous, or either, which again needs to be specified. Continuous actions are generally triggers that persists until a subsequent action. Creating complex rules

using only triggers is error prone since it is unlikely that two triggers will occur simultaneously. Thus our system will restrict trigger rules to containing a single trigger condition and possibly other non-triggers, and insist that trigger rules have a trigger action. Much, but not all of this, was implemented over time in the existing system.

B. Device Lessons.

Our previous implementation requires writing new code in the core system in order to add a new device. This is obvious impractical in a production environment. A more practical approach is to allow devices to be added and removed dynamically without requiring any code. This puts a little more burden on the device implementations. It also means we need to have devices that are known to the system but disabled so that rules defined for those devices don't disappear. Moreover, it can require a condition asking if a device is enabled or disabled.

To support multiple users and dynamic devices, our new approach is based on bridges to either specific devices or to device hubs. We provide a bridge to SmartThings that can include all the user's SmartThings devices; a bridge to a cloud-based implementation of our sign; a bridge to Google calendar; and a generic bridge that handles simple devices such as a monitor that determines if we are working at home. Such bridges require user credentials. Bridges can also be used to support devices such as weather access and events based on RSS-feeds that do not require credentials and can be shared among users.

A system that handles a large number of users and a larger number of devices can not afford to do polling of devices directly. This means that either the devices themselves should do the polling or the polling is the responsibility of the bridge for a particular user. The generic bridge in our new approach assumes that the devices are not connected continually, but rather ping the bridge occasionally and send any state changes to the bridge. The ping requests are used to send commands to the device and to request current device information.

Our existing system provides several devices aimed at determining the user's current location so that the sign can be updated accordingly. This includes checking if the user's phone is seen via Bluetooth from the office, a motion detector and an activity sensor. A better approach would be to assume the user carries their phone with them most of the time, and to have an app on the phone that determines its current location and can send it to the system. This would provide a more comprehensive and simpler approach that is more broadly applicable once privacy concerns are taken into account.

C. Debugging Lessons.

One of the most surprising aspects of our experience to date is the number of errors that occur when defining rules. About half of the time as new rules are defined incorrectly, so that either the new rule did not apply when it should because some other rule with similar conditions had higher priority, or it applied in cases where it should not have because it overrode the rule that should have applied. Fixing an erroneous rule was generally easy, involving either adding conditions or changing its priority relative to other rules. However, such problems should not occur in the first place and the user should not have to wait until they see the wrong output to fix the problem.

To counter this, we had the rule definition interface and the learning interface which could also be used for debugging. Neither of these proved particularly useful. The rule definition interface only showed rules that were overridden, and was not always accurate because of problems with the hypothetical worlds. The learning interface proved too cumbersome to use and did not provide accurate information about partial overrides.

Our new system will try to address this issue by providing a "Validate" button when defining a rule. This will analyze the current set of rules and provide the user with detailed information as to exactly when the rule would be applied, which rules might override it in other circumstances, and which rules the new rule override and under what circumstances. The current mechanism for doing this, hypothetical worlds, is not sufficient as it uses a discrete notion of time and does not understand either the effects of actions on properties of a device or the dependencies between conditions. We are hoping a comprehensive validation mechanism will obviate the need for actual after-the-fact debugging, and will make rule definition less error-prone.

D. User Interface Lessons.

While having multiple interfaces was helpful for understanding the domain, a practical system should be restricted to a single user interface. Moreover, this interface has to work with the limited screen space available on mobile devices while still offering enough of a global overview to make the user comfortable with using the system for multiple devices with a potentially large rule set.

Our strategy is to use the programmer's interface as the basis, but to incorporate modularity based solely on target device and a small set of higher-level priorities to restrict the number of displayed rules to a feasible few (5-7). Then to use the validation feature described above to handle debugging by ensuring that rules are defined correctly in the first place.

Finally, our current interface for defining what our sign should display is clumsy and needs to be simplified. Currently the sign is defined by the user using an SVG editor. This provides considerable flexibility but also requires considerable work on the user's part and typically requires a significant amount of screen space so it would be difficult to use on a mobile device. Defining the exact sign should not be part of the rule definition and needs to be simplified.

Our new approach separates the sign from the rest of the system. It uses a simple language for creating signs based on the set of signs we have created thus far. It lets user save and reuse directly or with modifications a set of standard signs. A rule for setting our new smart sign in our rule-based environment just provides the name of the target sign along

with optional information such as the end time of a meeting. This is more in line with how other devices are controlled.

IV. The Next Generation System

We are replacing our current system with a new system capable of handling multiple users and a large and varied set of devices. The system consists of the interacting components outlined below.

iQsign. We started with a cloud-based implementation of our smart sign designed to handle multiple users while making it simpler to define signs. This includes a textual language for defining signs with both text and images, and a library of saved signs and images. The back end and web interface is implemented in Node.js, with a Java system for actually generating sign images and a Flutter/Dart mobile app. It is accessible at https://sherpa.cs.brown.edu:3336.

Catre. Next we took our existing rule-interpreting back end and adapted it to handle our new requirements. This involved handling multiple users, running in the cloud rather than on one's personal machine, adding the notion of device bridges, and handling dynamic creation, disabling and removal of devices. It also involved simplifications such as removing hypothetical worlds and virtual devices, minimizing record keeping, and removing polling interfaces.

The system provides a RESTful interface designed to support an appropriate front end; various forms of user authentication both for the system and for the various bridges; and a more uniform approach to condition definition and management. Data is stored securely using MongoDB and the system is implemented in Java.

Cedes. This provides a multiple-user, web accessible bridge to various engines and devices. It communicates directly with Catre through a secure socket. It communicates to different engines such as SmartThings and iQsign over the Internet using user-supplied credentials. It also supports generic devices that periodically connect to either request information or to indicate changes to sensor values. This is implemented using Node.js.

Devices. This is an implementation of a set of devices that communicate using the Cedes generic bridge to provide some of the information we currently use to control our sign. It includes a monitor that runs on one's home machine to indicate if the user is currently active, whether they are in a zoom meeting, and whether they are in their personal zoom meeting. Current devices are implemented in Java.

ALDS. This is a mobile app that attempts to detect the user's location periodically using information available from GPS and from scanning for Bluetooth signals. It lets the user define a set of locations and specify when they are at a particular location. It then attempts to learn the characteristics of the different locations in order to automatically determine them later on. Note that locations do not have to be fixed; for example, "Driving" might be a location, as might "In Class" which could represent multiple classrooms. ALDS is implemented in Flutter/Dart.

Sherpa. This serves as the main user interface for using the overall framework. It attempts to implement much of what we learned from experience, providing a single interface with a fixed set of global priority levels and a smaller set of rules within each level, all modularized to a particular output device. It provides rule synopses on request. It provides a multiple screen approach to defining rules to account for limited screen space, handles differentiating trigger and continuous rules, and offers specialized interfaces for different types of conditions.

Sherpa also provides a rule validation mechanism for use while defining or editing rules. This mechanism will will detail the potential conflicts of the current rule in a clear and understandable manner. This mechanism will be invoked automatically when the user attempts to save a rule. Sherpa is being implemented as a mobile app using Flutter/Dart.

Evaluation. We will use the implemented system first to control the set of available smart devices in our house (thermostats, air purifier, lights, etc. and a smart sign) using various sensors and weather and calendar information. Once we feel the interface is practical for this purpose, we will run a controlled user study providing users with a fixed set of input and output devices and a number of tasks to set up. We will also make the system publicly available and solicit feedback from its users.

Future Work. Once the system is functional, we plan to look into using machine learning to help users create rules; continual improvement of the interfaces; and mechanism to automatically handle reconfiguration and adaption as in [10].

Source Code. All components of our next generation system are being developed open source and the source code is available on GitHub at https://github.com/stevenreiss/iot.

References

[1] IFTTT, "If this then that," IFTTT, http://ifttt.com, 2014.

[2] Samsung. (2023) Smartthings. [Online]. Available: https://developer.samsung.com/smartthings

[3] openHAB community. (2023) openhab: Welcome. [Online]. Available: https://www.openhab.org/docs/

[4] B. Ur, E. McManus, M. P. Y. Ho, and M. L. Littman, "Practical trigger-action programming in the smart home," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 803–812.

[5] B. Ur, M. P. Y. Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman, "Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, San Jose, California, USA, 2016, pp. 3227–3231.

[6] T. J.-J. Li, Y. Li, F. Chen, and B. A. Myers, "Programming iot devices by demonstration using mobile apps," in *International Symposium on End User Development (IS-EUD 2017)*, Jun. 2017, pp. 3–17.

[7] M. Kolberg, E. Magill, D. Marples, and S. Tsang, "Feature interactions in services for internet personal appliances," in *Proceedings of the IEEE International Conference on Softare Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2002, pp. 2613–2618.

[8] S. P. Reiss, "Iot end user programming models," in *2019 IEEE/ACM 1st International Workshop on Software Engineering Research and Practices for the Internet of Things (SERP4IoT)*, 2019, pp. 1–8.

[9] N. Labs, "Nest thermostat," Google, http://nest.com/thermostat, 2014.

[10] F. Durán, A. Krishna, M. Le Pallec, R. Mateescu, and G. Salaün, "Seamless reconfiguration of rule-based iot applications," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2021, pp. 142–148.