Iota: A Calculus for Internet of Things Automation

Julie L. Newcomb
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA, USA
newcombj@cs.washington.edu

Satish Chandra*
Samsung Research America
Mountain View, CA, USA
schandra@acm.org

Jean-Baptiste Jeannin†
Samsung Research America
Mountain View, CA, USA
jb.jeannin@gmail.com

Cole Schlesinger‡
Samsung Research America
Mountain View, CA, USA
cole@schlesinger.tech

Manu Sridharan§
Samsung Research America
Mountain View, CA, USA
manu@sridharan.net

Abstract

Programmatically controllable home devices are proliferating, ranging from lights, locks, and motion sensors to smart refrigerators, televisions, and cameras, giving end users unprecedented control over their environment. New domain-specific languages are emerging to supplant general purpose programming platforms as a means for end users to configure home automation. These languages, based on event-condition-action (ECA) rules, have an appealing simplicity. But programmatic control allows users write to programs with bugs, introducing the frustrations of software engineering with none of the tool support. The subtle semantics of the home automation domain—as well as the varying interfaces and implementation strategies that existing home automation platforms use—exacerbates the problem.

In this work, we present the Internet of Things Automation (Iota) calculus, the first calculus for the domain of home automation. Iota models an ECA language equipped with first-class notions of time, state, and device aggregation, and comes equipped with a precise semantics inspired by a careful analysis of five existing home automation platforms. We show that the Iota calculus is useful by implementing two analyses from the software engineering literature, and expressive by encoding sixteen programs from these home automation platforms. Along the way, we highlight how the design of the Iota semantics rules out subtle classes of bugs.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics →** *History of programming languages*;

Keywords domain-specific languages, language design and implementation

ACM Reference Format:
Julie L. Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. 2017. Iota: A Calculus for Internet of Things Automation. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3133850.3133860

1 Introduction

With the availability of cheap computing hardware [1] [2] and ubiquitous internet connectivity, it becomes technically feasible and commercially viable to connect home appliances—such as light bulbs, power outlets, motion detectors, ovens, refrigerators, and dishwashers—to inhabitants' smartphones and the Internet. These new connections give rise to the *Internet of Things* (IoT), enabling end users to not only remotely control their homes from their smartphones but also to automate certain everyday tasks. Such tasks can include turning on lights when users return home, notifying them when their children return from school, or watering the garden after sunset if it has not rained. Ideally, end users who want to automate and personalize their homes should be empowered to do so easily, efficiently, and playfully—even if they are not expert programmers.

This paper defines the Iota calculus, a core calculus for Internet of Things Automation that generalizes existing event-condition-action languages for home automation. The Iota

*This author is now at Facebook.
†This author is now at the University of Michigan - Ann Arbor.
‡This author is now at Barefoot Networks.
§This author is now at Uber.

ACM ISBN 978-1-4503-5530-8/17/10…$15.00
https://doi.org/10.1145/3133850.3133860

[1] http://www.raspberrypi.org/, accessed 2016-11-12.
[2] http://www.artik.io, accessed 2016-11-12.

calculus is an ECA language with first-class support for time, state, and device aggregation, important features for home automation applications. We define a precise semantics for the Iota calculus, use it to create useful tools for the end user, and show that programs written for several existing home automation platforms—including both ECA and general programming platforms—are naturally encoded in Iota.

A Spectrum of Programmability. The commercial market has flourished in recent years with home automation ecosystems from well-established companies and startups, as well as open-source solutions. One natural way to enable home automation is to equip a general-purpose programming language with an API for interacting with physical devices and appliances. But smart home applications are inherently reactive, concurrent shared-memory systems with complex timing constraints. For example, the Samsung SmartThings [3] tutorial teaches users to implement the following program:

> "When everyone leaves the house for more than ten minutes, turn the lights off."

It requires over one hundred lines of Groovy code, including sophisticated reasoning about locks, timers, interrupts, and mutable state. Not only is this tedious and error-prone, but general-purpose reactive programs are notoriously unreceptive to verification and tool support. The entire endeavor is also beyond the reach of end users without programming skills.

On the other end of the spectrum, some manufacturers offer a catalog of applications programmed and sold by third-party developers or the manufacturer itself, a model successfully pioneered on the smartphone market. However, every end user has a different home and different habits. As a result, downloadable apps tend to be too specialized or not offer enough personalization [20].

Simple event-condition-action (ECA) rule systems have emerged as a popular, practical means for non-technical users to configure and control home automation systems. The popular IFTTT service, Smart Lights for SmartThings, Muzzley [4], and Yonomi [5] are smartphone applications with simple interfaces that empower end users to install rules of (roughly) the following form:

> "**when** some event occurs
> **if** some condition is true
> **then** perform some action."

Such rule-based systems are easier to learn than general-purpose programming, while remaining customizable, striking an attractive middle ground. Although general purpose programming platforms clearly provide the most expressive power of the options above, for many home automation programs, this power is unnecessary. We surveyed home automation applications coded in general-purpose languages from app stores, user forms, and Github, and found that most are expressible in an event-condition-action rule system.

Need for Formal Analysis. Event-Condition-Actions frameworks strike a nice balance between expressivity and usability. Unfortunately, these systems are ad hoc: The expressivity of the rules varies, as do the semantics of execution. For example, three of the five home automation platforms we surveyed were prone to race conditions that arose needlessly from avoidable design decisions; a fourth exhibited deterministic but unpredictable behavior (see Section 4.2).

As a result, the experience of end users is often mixed. Writing a single rule is easy, but users can have trouble keeping track of all the rules in a larger program and create conflicting rules [13]. Since rules are declarative and not grouped in any larger structure, debugging a program when it behaves unexpectedly is difficult. A study by Brush *et al.* [3] of households using home automation found that users had difficulty writing programs that behaved as they expected: "Participants felt rules in general were hard to debug when they did not work and so participants lived with problems or turned off rules.... [Participant] also told us that he would have liked to try out different scenes if he knew how to experiment with the system."

Beyond a clear semantics, we believe tools for debugging and verifying home automation programs will be crucial in broadening adoption of home automation. Such tools can provide the user greater confidence that their programs behave as desired by detecting likely bugs early and enabling deeper diagnosis of past behaviors. Building such tools for general-purpose programming platforms is challenging, because home automation programs combine language features that are notoriously difficult to analyze, including concurrency, asynchronous callbacks, and shared mutable state. ECA languages are also appealing because their restricted nature makes them more amenable to analysis. However, some ECA languages that have emerged are *too* restricted to express common idioms, such as dealing with elapsed time.

Contributions. We present the Iota calculus which, to the best of our knowledge, is the first formal calculus for the domain of home automation. Iota comes equipped with an event-condition-action syntax and precise semantics. Iota is a formally defined ECA language with support for time, shared (but not allocatable) state, and device aggregation, while its semantics are carefully designed to avoid certain race conditions common to other systems. Iota is agnostic to the choice of platform and assumes only that home automation programs are capable of communicating with the devices they specify, and that all communication takes place via a central hub. We have implemented a simulator that can parse Iota programs and execute them in response to a sequence of events on an environmental configuration,

[3] http://www.smartthings.com, accessed 2016-11-11.
[4] http://www.muzzley.com, accessed 2016-11-11.
[5] http://www.yonomi.co, accessed 2016-11-11.

although we have not written bindings for actual hardware devices. IoT automations may be written directly in Iota by users with some familiarity with programming, and we feel its usability compares favorably with many of our comparison platforms. However, we expect that Iota will be most useful when used as the underpinnings for a higher level language or more sophisticated GUI.

We compare Iota with five existing home automation platforms: IFTTT, Smart Lights for SmartThings, Home Assistant, OpenHAB and SmartThings Apps. To evaluate Iota's expressiveness, we translate sixteen home automation programs drawn from the user communities of those five platforms. We compare the relative sizes of the original and encoded programs, and identify language features Iota lacks for encoding general-purpose home automation programs. We find Iota to be expressive and concise despite its restriction to a specific domain: Translated programs were often substantially smaller in Iota than their original languages.

We then use Iota and its formalism as a foundation for developing, implementing, and evaluating two analyses:

- *conflict detection* determines whether an event can trigger an execution that performs two conflicting actions;
- *positive and negative queries* determine the root causes of why events *did* or *did not* occur, tracing back through rule executions and intermediate events.

2 Current State of the Art

We chose five platforms as representative of the current approaches to home automation programming. These platforms serve as examples of various points in the spectrum between very restricted rule syntax to the power of a full general purpose programming language. All have nontrivial user bases and are available commercially or open source. We briefly review the features of each in order to demonstrate the ways in which home automation programs are currently written and to motivate our choices in the design of Iota and our analysis tools.

IFTTT [6] (If This Then That) is a service that connects APIs. Smart, cloud-connected devices publish services on IFTTT, providing *triggers* (events) which fire when an event occurs or condition is met, and *actions*, which update some field on the device. IFTTT rules simply connect one trigger to one action. They do not have conditions themselves; rather, conditions (and other arbitrarily complex logic) are baked into the manufacturer-provided triggers and actions.

SmartThings Smart Lights [7] allows users to create automated routines for IoT devices via a smartphone app. Using a GUI, users can write event-condition-action rules using the attributes exposed on registered devices. In addition to controlling devices, Smart Lights rules can transition between user-defined modes such as "away" or "normal business hours," as well as predicate on mode transitions. The platform also provides built-in predicates over domains such as time of day and the times of sunrise and sunset.

Home Assistant [8] is an open source platform written in Python. Users run the platform on a local server, often a Raspberry Pi. Bindings for a wide variety of devices are available, which expose sensors and fields that can be read and services that can control devices. Automation programs are made up of event-condition-action rules, written in YAML, a structured text format. Home Assistant rules closely resemble Iota rules, although they are implemented with a slightly different semantics. Home Assistant lacks explicit timers but can attach delays to event handlers (*e.g.* fire *m* minutes after an event has occurred) and to actions (*e.g.* in *m* minutes, perform some action). It can also set up triggers based on clock time using a crontab format.

OpenHAB [9] is another open source platform; similar to Home Assistant, it runs on a local server and provides bindings for a panoply of smart devices. Automation programs are written in Xtend [10], a Java-like expression language. Rules consist of two blocks, an event handler and an action block. Event handler blocks tend to be quite simple, but action blocks may contain arbitrary Xtend code and often contain control flow such as if statements. Xtend can use the Java API, including threading, locks, reflection, and other such features.

SmartThings SmartApps [11] Unlike the previously discussed platforms, SmartThings SmartApps are programs that can be published to an app store and that are intended to be written by developers, not end users. SmartApps are written in Groovy, a JVM language, and handle interactivity by attaching callbacks to events.

3 Iota by Example

The home automation programs found in each of these systems are concurrent, reactive programs with shared memory: They react to events from the world by modifying shared state, setting timers, and actuating devices. These actions may, in turn, raise new events.

As an example, consider a household with a few home automation devices: a porch light, a hallway light, a bedroom light, an automatic front door lock, and a motion detector on the porch. The lights can be switched on or off, and the bedroom light also has brightness and color settings. The door lock can be locked or unlocked manually or remotely.

[6] http://ifttt.com, accessed 2016-11-11.
[7] http://smartrulesapp.com, accessed 2016-11-11.
[8] https://home-assistant.io/, accessed 2016-11-11.
[9] http://www.openhab.org, accessed 2016-11-11.
[10] http://www.eclipse.org/xtend, accessed 2016-11-11.
[11] https://community.smartthings.com/t/list-of-all-officially-published-apps-from-the-more-category-of-smart-setup-in-the-mobile-app/13673, accessed: 2016-11-11.

Iota models device sensors and actuators as opaque names, such as hallway_light_switch. A light syntactic sugar groups device properties using record syntax, *e.g.* hallway_light.switch and hallway_light.brightness refer to the hallway light toggle and the brightness level.

Activating lights is a common task in the SmartThings community. The following Iota rule is built from an event handler, a predicate, and an action; it causes the hallway light to be turned on when the front door unlocks.

front_door.lock[locked $\hookrightarrow$];	*event handler*
front_door.lock = unlocked;	*condition*
hallway_light_switch := on	*action*

The first clause, front_door.lock[locked $\hookrightarrow$], is an event handler that matches events in which the "lock" property of the front door changes and the old value was "locked." The second clause is a predicate that must hold on the new state of the world in order for the action to fire—in this case, the event changes the door from locked to unlocked, and the new value of the lock must be unlocked. The final clause specifies the actions to be applied: here, we set the hallway light to on.

Perhaps we wish the light to turn off five minutes after the door locks.

```
front_door.lock[unlocked ↪];
front_door.lock = locked;
start light_timer at 0

light_timer[· ↪];
light_timer = 5;
hallway_light.switch := off, stop light_timer
```

These two rules together set and check a timer to enact a delayed action. If there exists no timer with the name light_timer, `start` light_timer `at` 0 will create one and start it; if one does exist, the action will reset its value to 0. Imagine a scenario in which a household member comes home (opening, closing, and locking the front door), then running outside briefly, and finally returning and locking the door again. Each time the front door is locked, the first rule fires, resetting the timer, and the light turns off five minutes after the last time the door was locked, as expected.

Finally, the following two rules implement the SmartThings "Bon Voyage" SmartApp: Turn off the hallway light when all family members' smartphones have been outside the house geofence for more than ten minutes. (As originally written, the SmartApp comprises 147 lines of Groovy code, including callbacks, interrupts, timers, and state.)

```
any {mom_phone, dad_phone, child_phone}
    (phone → phone.location[· ↪]);
all {mom_phone, dad_phone, child_phone}
    (phone → phone.location = away);
timer := 0

timer[· ↪ 10];
all {mom_phone, dad_phone, child_phone}
    (phone → phone.location = away);
hallway_light.switch := off
```

These rules make use of device aggregation: The any event handler ranges over a group of devices—family members' cell phones in this example—and matches if the location of any phone changes. The predicate `all` denotes the conjunction of a predicate lifted to a group of devices, here requiring that all locations be "away."

Although Iota is capable of expressing these and other home automation behaviors succinctly, it remains a core calculus, not a fully fledged language. As such, it excludes features like variable binding, arithmetic, and string manipulation that would be undeniably useful. We expect that adding these features would be straightforward.

4 Syntax and Semantics

Iota draws inspiration from the *trigger-action* programs of the ubiquitous computing community [20] and *event-condition-action* rules of active databases [5]. We begin by presenting the syntax of Iota, which comprises two levels: A base syntax of events, conditions, and actions with literal references to devices, and a surface syntax that accounts for device aggregation. After establishing the calculus syntax, we explore Iota's semantics, highlighting choices that have subtle implications for user experience and tool design.

4.1 Syntax

We begin with the simplest syntax $\mathcal{L}_1$ in Figure 1. Expressions range over device fields f, opaque timers m, and constant values n. The state of the system consists of a set of opaque fields, which hold the values of all readable and writable elements of the installed devices. While there is no memory allocation, internal state can be modeled as a finite number of virtual "device" fields. Timers are likewise virtual and, once started, are assumed to increment according to an internal clock.

An event is issued when the value of a field or timer changes, *e.g.* a light turns on. Events $f[n_1 \hookrightarrow n_2]$ are read as "the value of field f changes from n_1 to n_2." Event handlers match events; if the handler h of a rule matches an event e, then the rule is triggered. Handlers are written with the same notation as events but may omit the new value ($f[n \hookrightarrow]$) or the old *and* new values ($f[\cdot \hookrightarrow]$) to match regardless of the values of the event.

fields	f		
timers	m		
constants	n		
expressions	t	$::=$	$f \mid m \mid n$
event handlers	h	$::=$	$f[\cdot \hookrightarrow] \mid m[\cdot \hookrightarrow]$
		$\mid$	$f[n \hookrightarrow]$
		$\mid$	$m[n \hookrightarrow]$
events	e	$::=$	$f[n_1 \hookrightarrow n_2]$
predicates	p	$::=$	$\texttt{true} \mid \texttt{false}$
		$\mid$	$f_1 = f_2 \mid f = n \mid m = n$
		$\mid$	$f_1 < f_2 \mid f < n \mid m < n$
		$\mid$	$p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p$
actions	a	$::=$	$f := t \mid m := n \mid \texttt{stop}\ m$
action lists	as	$::=$	$\cdot \mid a, as$
rules	r	$::=$	$h; p; as$

Figure 1. Base syntax of IOTA.

devices	d	$::=$	*field set*
expressions	t	$::=$	$n \mid m \mid d.f \mid x$
groups	g	$::=$	$\texttt{devices} \mid \{d_1, \ldots, d_n\}$
		$\mid$	$(g \mid x \to p)$
event handlers	h	$::=$	$\ldots$
		$\mid$	$f[\cdot \hookrightarrow n]$
		$\mid$	$f[n_1 \hookrightarrow n_2]$
		$\mid$	$\texttt{any}\ g\ (x \to h)$
predicates	p	$::=$	$\ldots$
		$\mid$	$\texttt{all}\ g\ (x \to p)$
		$\mid$	$\texttt{exists}\ g\ (x \to p)$
		$\mid$	$f \in d$
actions	a	$::=$	$\ldots$
		$\mid$	$d.f := t$
		$\mid$	$\texttt{map}\ g\ (x \to a)$

Figure 2. IOTA extended with devices and aggregation.

Predicates form a Boolean algebra over field and timer tests. Fields may be compared with other fields and constants. Timers are based on the decidable fragment of timed automata and may only be compared with constant values [1]. Actions comprise field assignment and timer management: starting a timer at a particular value (`start` m `at` n) and stopping a timer (`stop` m).

Each rule is made up of an event handler, a predicate, and a list of actions. Note that an event algebra with disjunction can be encoded with multiple rules, one for each disjunct, and sequences of actions can be encoded by introducing intermediate state.

In $\mathcal{L}_1$, the program *"when the (previously closed) door opens, turn the light on"* can be written as

$$\text{door}[\text{closed} \hookrightarrow \text{open}]; \texttt{true}; \text{lightswitch} := \text{on}$$

where door and lightswitch are fields that encode devices, and closed, open, and on are constants.

Device Aggregation. Devices are collections of sensors, actuators, and metadata, and it is often convenient to refer to groups of devices sharing a common attribute. We extend the syntax of $\mathcal{L}_1$ to capture devices, their properties, and operators that range over groups of devices: We say that $\mathcal{L}_2$ comprises $\mathcal{L}_1$ extended with the constructs presented in Figure 2. We review the syntactic sugar of $\mathcal{L}_2$ informally below; the formal desugaring rules are given in the appendix.

Devices d are now collections of fields, and fields in expressions are replaced by field projection over devices. Variables x also appear in expressions. Groups may be an explicit set of devices or a special keyword `devices`, which denotes all devices registered with the system. Groups may be further refined using the construct $(g \mid x \to p)$, which filters a group g using an anonymous function with x ranging over devices in g and binding in p. Events are extended with the existential construct `any`, predicates with universal (`all`) and existential (`exists`) constructs, and actions with `map`, which applies an action to a group of devices. Predicates also include a test $f \in d$, which checks whether a field is present on a device.

$\mathcal{L}_2$ introduces a concise way to turn all the lights on when any door opens:[12]

$$\begin{array}{l}\texttt{any devices}\ (x \to x.\text{door}[\text{closed} \hookrightarrow]);\\ \texttt{true};\\ \texttt{map devices}\ (x \to x.\text{switch} := \text{on})\end{array}$$

Finally, we extend event handlers to specify the value the field holds after it is updated, which would otherwise need to be checked in the rule predicate.

At present, IOTA is untyped. However, one could imagine augmenting the calculus with a simple type system to distinguish command domains, such as *on, off* for a light switch compared to *heat, cool, off* for an A/C unit. We hope to explore this and other type system features in future work.

4.2 Semantics Design

IOTA assumes that all control logic takes place on a central hub, as do all the platforms we survey, whether that hub is a server in the cloud or a Raspberry Pi in the home. Communication between devices and the hub takes place via wifi, Bluetooth, or some other wireless protocol; thus, network latency is a major factor in the execution of home automation programs. In designing semantics for IOTA, we were careful to create strong guarantees about evaluations that take place on the hub, and to avoid making guarantees about the timing or sequence of events that occur via communication to and from the distributed array of IoT devices.

When an event occurs, the system hub is notified of the change and calculates a response, which is dispatched to the relevant devices as updates. We found that three of the five

[12] Projecting a nonexistent field from a device is silently ignored.

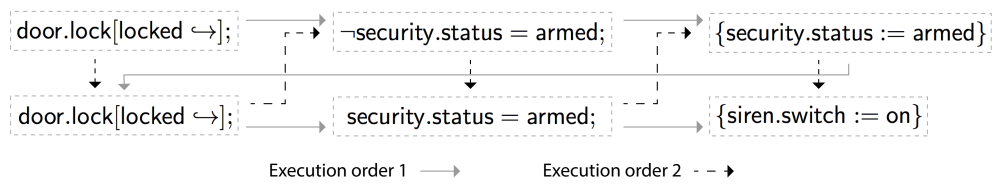

Figure 3. Different execution models for rules in response to a single event.

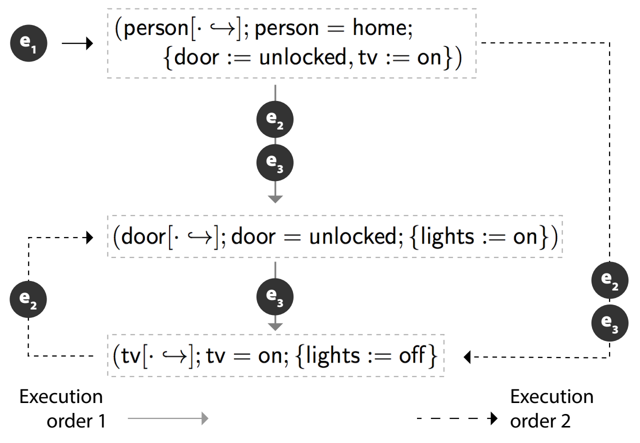
e1
(person[· ↪]; person = home; {door := unlocked, tv := on})
e2
e3
(door[· ↪]; door = unlocked; {lights := on})
e2
e3
e2
e3
(tv[· ↪]; tv = on; {lights := off})
Execution order 1
Execution order 2

Figure 4. Nondeterministic execution arising from evaluating one event at a time from a set of events.

systems examined suffered from race conditions in response to a single input event; a fourth, while deterministic, obeys a logic that is opaque to the user.

As an example, consider the two rules in Figure 3, which arm a security system when a user returns home and sound an alarm if someone (presumably an intruder) tries to enter the house thereafter.[13] Suppose the door changes from *locked* to *unlocked* and the security system is not armed. How should the system behave?

One option is to evaluate each rule one at a time. Both rules in this scenario are triggered by this event, so both orderings are valid. In the first ordering (*Execution order 1* in Figure 3) , the first rule would check that the security system is not armed, and upon finding that it is not, would then arm it. Next, the second rule will be evaluated, but since the security system is now armed, the actions on the second rule will fire, immediately activating the siren. Thus, depending on how the race is resolved, the siren could go off any time anyone came home. Another option, which is simple to implement, would be to spawn a new thread to handle each rule that matches an event (not shown in Figure 3). This, of course, would lead to similar race conditions.

Indeed, Zave *et al.* observe that home automation features (collections of rules) often overlap, and suggest assigning priorities to determine which feature takes precendence in the event of interaction [22]. For example, if two rules are triggered and set the thermostat to two different temperatures, the rule with higher priority will be the one to take effect. However, it seems unlikely that users will be able to internalize the relative priority of tens or hundreds of rules; it would also be difficult to recollect priorities days or months later when updating the automation program.

[13]This was drawn from an actual user program.

None of our five comparison platforms use rule priorities. We experimented with each platform with respect to the example above to determine whether such race conditions are possible. This type of race condition cannot occur in IFTTT, as IFTTT rules have no conditions. The SmartThings Smart Lights app evaluates rules one at a time in an arbitrary but fixed order. The order seems to be determined by the creation date of each rule, but this is not made clear to the programmer. Although this does not yield race conditions during execution, adding or removing rules can change the behavior of the system unexpectedly.

The race condition is possible in both Home Assistant and OpenHAB; both platforms fire asynchronous actions when rules are fully evaluated and the resulting undetermined interleaving of actions can lead to a race. OpenHAB rules do not have separate condition statements, but their action blocks can be quite complex, containing if statements or other forms of control flow, and untangling the interactions between rules can be difficult. Finally, while a well-written SmartApp should attach one callback per event, nothing prevents two SmartApps from responding to the same event and causing this type of error. Nor are SmartApps, which run in a threaded environment, guaranteed to finish handling one event before responding to another.

The Iota semantics evaluates rules in phases, which eliminates this class of race condition without resorting to rule prioritization. First, all the event handlers are evaluated, then predicates, and finally actions. *Execution order 2* in Figure 3 illustrates this behavior: By evaluating all predicates before applying any actions, the alarm only fires when the door is unlocked a second time, as intended.

Moreover, all actions are evaluated against the same initial state; the effects of one action do not influence others when handling the same event. Hence, event handlers and predicates are commutative, as are actions that do not write to the same state.

After an action is sent to its device as an update, upon completion the device will notify the hub of a new event,

$$
\begin{array}{lcl}
\multicolumn{3}{l}{[\![\cdot]\!]_h :: \mathit{Rule} \rightarrow \mathit{Event} \rightarrow \mathit{Rule\ Set}} \\
[\![(f[\cdot \hookrightarrow]; p; a)]\!]_h\ f[n \hookrightarrow n'] & = & \{(f[\cdot \hookrightarrow]; p; a)\} \\
[\![(f[n \hookrightarrow]; p; a)]\!]_h\ f[n \hookrightarrow n'] & = & \{(f[n \hookrightarrow]; p; a)\} \\
[\![(h; p; a)]\!]_h\ f[n \hookrightarrow n'] & = & \emptyset \text{ otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{[\![\cdot]\!]_p :: \mathit{Rule} \rightarrow \mathit{State} \rightarrow \mathit{Rule\ Set}} \\
[\![(h; \texttt{true}; a)]\!]_p\ \Sigma & = & \{(h; \texttt{true}; a)\} \\
[\![(h; \texttt{false}; a)]\!]_p\ \Sigma & = & \emptyset \\
[\![(h; x_1 \oplus x_2; a)]\!]_p\ \Sigma & = & \{(h; x_1 \oplus x_2; a)\} \\
\quad \text{when } x_1 \oplus x_2 \in \{f_1 \oplus f_2, f \oplus n, m \oplus n\} & & \\
\quad \text{and } \Sigma[f_1] \oplus \Sigma[f_2] & & \\
 & = & \emptyset \text{ otherwise} \\
[\![(h; p_1 \wedge p_2; a)]\!]_p\ \Sigma & = & \{(h; p_1 \wedge p_2; a)\} \\
\quad \text{when } [\![(h; p_1; a)]\!]_p\ \Sigma \wedge [\![(h; p_2; a)]\!]_p\ \Sigma & & \\
 & = & \emptyset \text{ otherwise} \\
[\![(h; p_1 \vee p_2; a)]\!]_p\ \Sigma & = & \{(h; p_1 \vee p_2; a)\} \\
\quad \text{when } [\![(h; p_1; a)]\!]_p\ \Sigma \vee [\![(h; p_2; a)]\!]_p\ \Sigma & & \\
 & = & \emptyset \text{ otherwise} \\
[\![(h; \neg p; a)]\!]_p\ \Sigma & = & \{(h; \neg p; a)\} \\
\quad \text{when } [\![(h; p; a)]\!]_p\ \Sigma = \emptyset & & \\
 & = & \emptyset \text{ otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{[\![\cdot]\!]_a :: \mathit{Rule} \rightarrow \mathit{State} \rightarrow \mathit{State} \times \mathit{Event}} \\
[\![(h; p; f := n)]\!]_a\ \Sigma & = & \Sigma[f \rightarrow n], f[\Sigma[f] \hookrightarrow n] \\
[\![(h; p; f_1 := f_2)]\!]_a\ \Sigma & = & \Sigma[f_1 \rightarrow (\Sigma[f_2])], \\
 & & f[\Sigma[f_1] \hookrightarrow \Sigma[f_2]] \\
[\![(h; p; m := n)]\!]_a\ \Sigma & = & \Sigma[m \rightarrow n], m[\Sigma[m] \hookrightarrow n]
\end{array}
$$

Figure 5. Denotational semantics for the evaluation of event handlers, predicates, and actions.

possibly triggering another rule. Iota makes no assumptions about the order in which these new events will be received and evaluated; in our semantics, any ordering of event processing is valid. The design of the semantics thus does not preclude write-write conflicts. Rather than attempt to prevent them (tricky in a distributed system), we develop a conflict detection algorithm to identify them (Section 6).

Since actions from the execution of one rule can trigger the execution of another, it would be possible to design a semantics in which the entire tree of rule executions is evaluated on the central hub, with all updates being dispatched to devices after processing. This would avoid the latency required for communication between devices and the hub, and could prevent some types of conflict. However, due to network connections over wifi or Bluetooth, battery failures, misconfigurations, or other issues, it is likely that a network of IoT devices will not be perfectly robust. Instead, we choose to send an update and wait for acknowledgement before executing any rule that update might trigger. We thus avoid the case in which an update fails, but we continue with execution as if it had succeeded, which might result in unpredictable outcomes.

4.3 Evaluation Semantics by Example

We develop a denotational semantics to model the behavior of individual event handlers, predicates, and actions, which execute atomically (by design), and a small-step semantics to capture the interleaving behavior of rule application over a set of events. Given a set of rules, an event, and the current state of the world, all event handlers are evaluated, followed by all predicates, and then all actions. The order in which individual event handlers, predicates, and actions are applied is not specified; any ordering is legal.

When an action is evaluated and an update is successfully written to the corresponding device, an event corresponding to the update is also generated, causing another round of rule evaluation. If a rule causes more than one update, the order in which the new events are evaluated is unconstrained. Figure 4 illustrates this case: Event e_1 matches the first rule, which in turn creates two new events: one for the door unlocking, and another for the television turning on. Either of the new events could be chosen for evaluation next.

The remainder of this section illustrates evaluation by example. The full semantics of rule set evaluation are given in the appendix.

Consider a rule that turns off the heater at 8:00 if no one is at home; an initial state Σ in which the heater is on, no one is home, and the time is 7:59; and an event in which the clock changes from 7:59 to 8:00.

Event handlers are modeled as functions from rules and events to sets of rules. They act as filters, producing a singleton set containing the original rule if the event triggers the rule and the empty set otherwise. In our example, we see the evaluation of the event handler triggered as the clock changes to 8:00.

$$
\begin{array}{l}
[\![(\mathit{clock.hour}[7 \hookrightarrow 8]; p; a)]\!]_h\ \mathit{clock.hour}[7 \hookrightarrow 8] = \\
\quad \{(\mathit{clock.hour}[7 \hookrightarrow 8]; p; a)\}
\end{array}
$$

Likewise, given a rule and a state, predicates evaluate to a set that contains the original rule if it the predicate holds on the state and to the empty set otherwise.

$$
\begin{array}{l}
[\![(h; \mathit{household.status} = \mathit{away}; a)]\!]_p \Sigma = \\
\{(h; \mathit{household.status} = \mathit{away}; a)\}
\end{array}
$$

Finally, actions are modeled as functions from a rule and state to a new state, in which the actions of the rule have been applied, and a set of events that arise as a result of those actions.

$$
\begin{array}{l}
[\![(h; p; \mathit{heater.status} := \mathit{off})]\!]_a \Sigma = \\
\Sigma[\mathit{heater.status} \rightarrow \mathit{off}], \mathit{heater}[\mathit{on} \hookrightarrow \mathit{off}]
\end{array}
$$

With the behavior of these core elements established, we turn to a small-step semantics to define how sets of events are processed by sets of rules. Formally, an event e_1 (the event being processed), a set of events es_1 (the remaining events), a state Σ_1, and an evaluation state s step to $(e_2, es_2, \Sigma_2, s_2)$,

$$\begin{array}{llll} \text{evaluation state} & s & ::= & \textbf{none}\ (rs) \\ & & \mid & \textbf{event}\ (rs_{in}, rs_{out}) \\ & & \mid & \textbf{pred}\ (rs_{in}, rs_{out}) \\ & & \mid & \textbf{act}\ (rs_{in}, es_{out}) \end{array}$$

$$\boxed{e_1, es_1, \Sigma_1, s_1 \xrightarrow{rs} e_2, es_2, \Sigma_2, s_2} \qquad \frac{}{\cdot, e \uplus es, \Sigma, \textbf{none}\ (rs) \xrightarrow{rs} e, es, \Sigma, \textbf{event}\ (rs, \emptyset)}\ \text{R-E}$$

$$\frac{[\![r]\!]_h\ e = rs}{e, es, \Sigma, \textbf{event}\ (\{r\} \uplus rs_1, rs_2) \xrightarrow{rs} e, es, \Sigma, \textbf{event}\ (rs_1, rs \cup rs_2)}\ \text{R-H} \qquad \frac{}{e, es, \Sigma, \textbf{event}\ (\emptyset, rs) \xrightarrow{rs} e, es, \Sigma, \textbf{pred}\ (rs, \emptyset)}\ \text{R-HP}$$

$$\frac{[\![r]\!]_p\ \Sigma = rs}{e, es, \Sigma, \textbf{pred}\ (\{r\} \uplus rs_2, rs_3) \xrightarrow{rs} e, es, \Sigma, \textbf{pred}\ (rs_2, rs \cup rs_3)}\ \text{R-P} \qquad \frac{}{e, es, \Sigma, \textbf{pred}\ (\emptyset, rs) \xrightarrow{rs} e, es, \Sigma, \textbf{act}\ (rs, \emptyset)}\ \text{R-PA}$$

$$\frac{[\![r]\!]_a\ \Sigma = \Sigma_2, es_2}{e, es, \Sigma, \textbf{act}\ (\{r\} \uplus rs_3, rs_4) \xrightarrow{rs} e, es \cup es_2, \Sigma_2, \textbf{act}\ (rs_3, \{r\} \cup rs_4)}\ \text{R-A} \qquad \frac{}{e, es_1, \Sigma, \textbf{act}\ (\emptyset, rs_4) \xrightarrow{rs} \cdot, es \cup es_2, \Sigma, \textbf{none}\ (rs)}\ \text{R-AE}$$

Figure 6. Semantics of ruleset evaluation

written

$$(e_1, es_1, \Sigma_1, s_1) \xrightarrow{rs} (e_2, es_2, \Sigma_2, s_2)$$

Evaluation begins with a set of events and the evaluation state **none** (rs). The first transition nondeterministically selects an event to process and moves to the **event** $(rs, \cdot)$ phase, indicating that the event handlers of rules rs must be evaluated against the event e.

$$(\cdot, e \uplus es, \Sigma, \textbf{none}\ (rs)) \xrightarrow{rs} (e, es, \Sigma, \textbf{event}\ (rs, \cdot))$$

Each subsequent step selects a rule in rs, evaluates its event handler against e, and moves it to the right side of the event state if it triggers. Eventually, all the rules will be evaluated, and we transition to the next phase of evaluation: predicates.

$$(e, es, \Sigma, \textbf{event}\ (\cdot, rs')) \xrightarrow{rs} (e, es, \Sigma, \textbf{pred}\ (rs', \cdot))$$

Hence, the event handlers, predicates, and actions are processed in batches. Once the final action has been evaluated, the event is discarded and a new event selected from es, until no events remain.

In our example, the first step chooses an event to begin processing—here there is only one event to choose from, but if there were more, any choice of an event to process would be valid. This reflects the fact that new updates may arrive from distributed devices in any order. For brevity, we let r stand for the rule.

$$\frac{}{\begin{array}{c}(\cdot, clock.hour[7 \hookrightarrow 8] \uplus \emptyset, \Sigma, \textbf{none}\ (\{r\})) \xrightarrow{rs} \\ (clock.hour[7 \hookrightarrow 8], \emptyset, \Sigma, \textbf{event}\ (\{r\}, \emptyset))\end{array}}\ \text{R-E}$$

From there, we choose a rule to match its event handler against the event. Although there is only one rule in our example, note that rules are in an unordered set and at any step, choosing any rule from the appropriate set is valid. The order rules are processed in the event and predicate evaluation states does not affect the final result.

$$\frac{[\![r]\!]_h\ e \quad = \{r\}}{\begin{array}{c}(clock.hour[7 \hookrightarrow 8], \emptyset, \Sigma, \textbf{event}\ (\{r\}, \emptyset)) \xrightarrow{rs} \\ (clock.hour[7 \hookrightarrow 8], \emptyset, \Sigma, \textbf{event}\ (\emptyset, \{r\}))\end{array}}\ \text{R-H}$$

Once all rules have been processed in the event evaluation state, we step to the pred evaluation state.

$$\frac{}{\begin{array}{c}(clock.hour[7 \hookrightarrow 8], \emptyset, \Sigma, \textbf{event}\ (\emptyset, \{r\})) \xrightarrow{rs} \\ (clock.hour[7 \hookrightarrow 8], \emptyset, \Sigma, \textbf{pred}\ (\{r\}, \emptyset))\end{array}}\ \text{R-HP}$$

We step through the pred evaluation state in a similar fashion. When we reach the act evaluation state, we take the sole rule and apply the action step, which produces an updated state Σ_2, in which the *heater.status* is now off, and issues a new event, *heater.status*[*on* ↪ *off*]. Like the other two evaluation state phases, the order of rule processing is not constrained, although it can affect the final state of the system in the case of write-write conflicts.

$$\frac{[\![r]\!]_a\ \Sigma = \Sigma_2, \{heater.status[on \hookrightarrow off]\}}{\begin{array}{c}(clock.hour[7 \hookrightarrow 8], \emptyset, \Sigma, \textbf{act}\ (\{r\}, \emptyset)) \xrightarrow{rs} \\ (clock.hour[7 \hookrightarrow 8], \{heater.status[on \hookrightarrow off]\}, \\ \Sigma_2, \textbf{act}\ (\emptyset, \{r\}))\end{array}}\ \text{R-A}$$

Now that all rules have been fully processed, we finish by stepping into the none evaluation state. From here we can continue by selecting another event to process from the set of pending events.

$$\frac{}{\begin{array}{c}(clock.hour[7 \hookrightarrow 8], \{heater.status[on \hookrightarrow off]\}, \\ \Sigma_2, \mathbf{act}\ (\emptyset, \{r\})) \xrightarrow{rs} \\ (\cdot, \{heater.status[on \hookrightarrow off]\}, \Sigma_2, \mathbf{none}\ (\{r\}))\end{array}} \text{R-AE}$$

5 Evaluation of Iota Expressiveness

We evaluate the expressiveness of Iota on programs of interest to end users by gathering sixteen programs from public forums associated with each platform and translating them into Iota programs. For programs beyond the reach of Iota, we note the language feature(s) that Iota does not capture. Table 1 summarizes our results. We found that programs from existing ECA languages were easily translated to Iota, and larger programs from general purpose languages translated to much smaller Iota programs. We discovered that language features missing from Iota were largely as expected for a calculus: Missing base types and associated operations, like integers and arithmetic. Notably, these programs did not make use of looping constructs or state allocation, which would be difficult to model in Iota.

IFTTT. From the programs published on the IFTTT website, we chose five that use common IoT devices, such as automatic blinds, a rain gauge, and a smart thermostat. As each IFTTT program has exactly one event handler and one action, all five programs translated cleanly into Iota. This simplicity results in a limited amount of expressive power; only about half of the Smart Lights programs we examined could be expressed by IFTTT, and only a small fraction of the Home Assistant and OpenHAB programs.

One thing to note is that although the IFTTT rules themselves are very simple, more intelligence can be built into the trigger and action channels. For example, one program uses a trigger sent by a smart refrigerator if its door has been open for over a minute, even though IFTTT has no explicit timer support. Although channel implementation is not part of IFTTT—and, indeed, is likely out of reach for many end users—we encoded this channel behavior in Iota with three rules: one that sets a timer when the refrigerator door is opened, one that cancels the timer when the door is closed, and one that sends a message if the timer reaches 60 seconds.

SmartThings Smart Lights. We were unable to find publicly available Smart Lights programs, so we translated five "user story" blog posts from the Smart Lights website into Iota programs. These programs include temperature and moisture sensors that control a greenhouse; geofenced phones that control lighting for family members; and an office with different behaviors during and outside normal office hours. Each was straightforward to encode in Iota. The Smart Lights programs maintain "modes", named fields that are not associated with any physical device. Rules are used to calculate properties such as "household is home" or "normal office hours," which then drive other rules. Iota naturally captures this behavior with named state fields. Some Smart Lights programs also show the utility of Iota's "group" syntactic sugar, which can express predicates such as "any household member's phone enters the home geofence area" and "update the state to 'home' if all household members are at home" with a single rule.

Home Assistant. For Home Assistant and OpenHAB, we chose two large programs that were written for a household equipped with many tens of IoT devices. The Home Assistant program[14] has 132 rules, controlling over 10 types of devices as well as many non-device properties. In translating the sample program we found no feature that could not be expressed in Iota. Home Assistant automation rules are the closest in syntax to Iota of the five platforms we surveyed, and for the most part Home Assistant rules translate one-to-one to Iota rules. The exception is rules governing timers: Home Assistant lacks full support for timers, but time delays can optionally be added to event handlers; for example, a trigger can be written that will fire if a motion detector does not sense motion for three minutes. Writing the same rule in Iota requires explicitly managing a timer, which is done with three rules: one to set the timer, one to perform an action when the timer reaches the desired time, and one to cancel the timer if necessary. However, adding a delay construct to Iota as syntactic sugar is straightforward.

Both Home Assistant and Iota allow event handlers, predicates, and actions to be written over groups of devices. However, neither Home Assistant nor Iota capture parameters; for example, neither can write a rule that states when any member of the household leaves home, their personal "away" property becomes true. Both platforms would require a separate rule for each person.

OpenHAB. The OpenHAB program[15] consists of 103 rules, managing devices such as lights, window shades, heating and cooling, motion detectors, temperature and humidity sensors, time and date, and so on. OpenHAB rules consist of an event handler and an action block; since the action block can contain arbitrary code, it is much more expressive than Iota. However, the only inexpressible behavior in the sample program was arithmetic (primarily for counting and calculating temperatures) and string manipulation (for logging error reports). More sophisticated language constructs, such as loops, exceptions, interrupts, and locks—which would prove difficult to model in Iota—were not present. We do not anticipate difficulty in extending Iota with arithmetic or string manipulation.

Note that Iota required about 30% more rules than OpenHAB to express equivalent sample programs. This is largely because of two factors: first, Iota event handlers cannot aggregate over devices with different properties, requiring the

[14] https://github.com/geekofweek/homeassistant/, accessed 2016-11-11.
[15] http://www.intranet-of-things.com/software/downloads, accessed 2016-11-11.

single OpenHAB rule to be split into two Iota rules, as seen in Figure 7. Second, each Iota rule executes a specified set of actions if its predicate evaluates to true, while OpenHAB rules may contain several sets of actions guarded by if statements. Every set of truth conditions that would result in a distinct set of actions executed requires a separate rule in Iota. Although more verbose, the simpler structure of Iota rules is more amenable to analysis.

SmartThings SmartApps. We translated three SmartThings SmartApps in addition to the "Bon Voyage" example presented in Section 1: one to manage heating in response to thermostat settings; one to control lighting depending on motion detectors, light sensors, and sunrise/sunset times; and one to set thermostat settings based on weather information. Each program contained hundreds of lines of code, but much of it consisted of boilerplate code for specifying which devices to manipulate and configuring user options, both of which are explicit in Iota rules.

However, we also encountered one feature that Iota was incapable of expressing concisely. The Gidjit SmartApp is essentially a shim that connects SmartThings devices with a remote service—it receives RESTful messages (which we encode as a device that generates an event when a message is received) and uses the contents of its messages to refer to particular devices. Reflecting device names is technically possible to encode in Iota by enumerating all possible message literals and connecting them to the devices named therein, but such an encoding is very inefficient. Since reflection is such a powerful feature, it limits the types of analysis we can perform.

```
rule "Christmas lamps on"
when
  Time cron "0 0 16 * * ?"
  or
  Item State_Sleeping changed from ON to OFF
then
  if( Auto_Christmas.state == ON &&
    ( Socket_Livingroom.state == OFF ||
    Socket_Floor.state == OFF ) ) {
    var Number hour   = now.getHourOfDay
    var Number minute = now.getMinuteOfHour

    if( ( hour == 16 && minute == 0 ) ||
     ( hour < 10 && State_Sleeping.state == OFF
      && State_Away.state == OFF ) ) {
      sendCommand(Socket_Livingroom,ON)
      sendCommand(Socket_Floor,ON)
    }
  }
end
```

clock.hour[$\cdot \hookrightarrow$ 16];
state.auto_christmas = on
$\wedge$exists livingroom, floor ($x \rightarrow x$.socket = off);
map livingroom, floor ($x \rightarrow x$.socket := on)

state.sleeping[$\cdot \hookrightarrow$ off];
state.auto_christmas = on
$\wedge$clock.hour < 10$\wedge$
exists livingroom, floor ($x \rightarrow x$.socket = off);
map livingroom, floor ($x \rightarrow x$.socket := on)

Figure 7. An OpenHAB rule translated into Iota. At 4pm, if Christmas mode is enabled, turn the living room and floor sockets on. When the household wakes up, if it is before 10am and Christmas mode is enabled, turn the living room and floor sockets on.

5.1 Features Not Addressed with Iota

Security and privacy are very important concerns for the Internet of Things. Although we do not address it directly in this work, our belief is that by clearly defining a core calculus with well-established semantics, future work will be able to provide better security analysis than is possible with current IoT platforms. Likewise, providing end users with a well-considered programming language with good tool support will enable them to better understand and control their home automations, which can only improve security.

When the components of a home automation system are first installed, or when new devices are added to an existing system, they must be registered to communicate with the center hub. This may include a form of authentication, arranging for secure communication, and so on. We believe device onboarding to be relatively infrequent, occurring when a user purchases new devices or offers trusted guests access to the system. The mechanisms for device onboarding are beyond the scope of this work.

Finally, we assume that all devices that are connected to Iota have bindings to send information about their current state, to alert a central hub of changes, and to receive instructions to change their status. Such bindings will need to be defined per device, which is not addressed in this work.

6 Analyses

Iota, with its limited syntax and clearly defined semantics, is highly amenable to analysis. Here, we present two analyses for Iota that have been applied successfully in other domains: conflict detection, based on an application of bounded model checking [2], and a provenance analysis, inspired by the Whyline work on program understanding [8]. Beyond demonstrating the analyzability of Iota, we believe these two tools would be directly valuable to an end user writing home automation programs.

Table 1. Comparison of platform programs.

Platform	Programs translated	Average lines/ rules per program	Average number of IOTA rules per program	Features missing from IOTA
IFTTT	5	1 rule	1	None
Smart Lights	5 rules	–	8	None
Home Assistant	1	132 rules	162	None
OpenHAB	1	103 rules, 2085 LOC	136	Arithmetic; string manipulation
SmartApps	4	324 LOC	8	Reflection

6.1 Conflict Detection

One technique to check for bugs in a ruleset is to determine if any rules contradict each other, or are in *conflict*. We consider a set of rules to be in conflict if there is some initial state from which a single input event can trigger a series of actions in which some field is written to more than once. According to IOTA semantics, evaluating this event with the starting state against the conflicting ruleset would result in an unpredictable final state. In practice, one cause of nondeterminism is network latency, which often dominates computation time in home automation systems. The actions produced by a conflicting execution (for example, a light turning on and then turning off) may be reordered on delivery to the light. Even if delivered in order, behavior can be highly dependent on the device: The light may flicker, turn on and then off with an arbitrary delay, or never turn on at all. Indeed, we observed each of these behaviors experimenting with the Muzzley, Yonomi, and SmartThings systems connected to a Philips Hue light. Even actions that send the same value to a field may be a problem if they are expensive or have side effects (for example, sending a text message).

Example Rule Sets with Conflicts. Consider the following example.

$$\mathsf{person}[\cdot \hookrightarrow \mathsf{home}]; \mathtt{true}; \{\mathsf{lights} := \mathsf{on}\}$$
$$\mathsf{person}[\mathsf{garage} \hookrightarrow]; \mathtt{true}; \{\mathsf{lights} := \mathsf{off}\}$$

According to the first rule, when a person (as located by, say, a cell phone) arrives home, IOTA should turn the lights on. But, according to the second rule, when they leave the garage, IOTA should turn the lights off. If a person were to leave the garage and enter the house, a conflict would result.

In the next example, the conflict arises because the first rule fires two events, which are evaluated in arbitrary order. After the event $person[away \hookrightarrow home]$ is fully evaluated, we cannot predict if the lights will be on or off.

$$\mathsf{person}[\cdot \hookrightarrow \mathsf{home}]; \mathtt{true}; \{\mathsf{door} := \mathsf{unlocked}, \mathsf{tv} := \mathsf{on}\}$$
$$\mathsf{door}[\cdot \hookrightarrow \mathsf{unlocked}]; \mathtt{true}; \{\mathsf{lights} := \mathsf{on}\}$$
$$\mathsf{tv}[\cdot \hookrightarrow \mathsf{on}]; \mathtt{true}; \{\mathsf{lights} := \mathsf{off}\}$$

Finally, consider the following deterministic but still problematic example.

$$\mathsf{person}[\mathsf{home} \hookrightarrow]; \mathtt{true}; \{\mathsf{door} := \mathsf{locked}, \mathsf{lights} := \mathsf{off}\}$$
$$\mathsf{door}[\cdot \hookrightarrow]; \mathtt{true}; \{\mathsf{lights} := \mathsf{on}\}$$

When the person leaves home, the first rule locks the door and turns off the lights, but when the door's status changes, the lights will turn on. In this case, the lights will turn off and immediately back on. Likely the user intended that the lights turn on only when the door is *unlocked*. In the next section, we explore how the user can diagnose other forms of unexpected behavior.

Conflict Detection through Bounded Model Checking. Our implementation of conflict detection is based on Spin, an open-source verification tool for bounded model checking [16]. We chose Spin because of its simple syntax and good support of nondeterminism. Due to IOTA's limited expressiveness and explicit semantics, we were able to write a simple transpiler to convert any IOTA program into a semantically equivalent Spin model. The end user need not understand or even see the Spin model itself to use this tool.

In our implementation, we first declare an environment holding all device fields and their possible values. Each rule is then translated into a Spin method that updates the environment with any triggered rule actions. Events caused by those actions trigger other rules in turn, in any order permitted by the IOTA semantics. Finally, we add a counter for every writable field in the environment and increment it every time an action updates that field. We then assert that at the end of execution, every counter must have a value of 1 or less. The model checker looks for any initial environmental configuration, any single world event, and any order of subsequent rule execution that would result in some field being updated more than once. If such an execution can be found, the ruleset contains a conflict.

A few optimizations are necessary to make bounded model checking practical. In IOTA, fields can hold enumerations or integers. To limit the state space, we derive a set of possible values for each integer field by finding all predicates that reference that field, constructing a truth table from those predicates, and using the Z3 solver [17] to find concrete values for each entry in the truth table, excluding any entry that is unsatisfiable. For example, given a rule with the predicate $\mathsf{temperature} < 30$, we construct a truth table with two entries, one where the predicate evaluates to true and

[16] http://spinroot.com/spin/whatispin.html, accessed 2016-11-14.
[17] https://github.com/Z3Prover/z3, accessed 2016-11-15.

one where it evaluates to false. We then call Z3 to find concrete values for each entry, perhaps 15 and 45. We can thus limit the possible values for temperature to those two values when exploring all possible states. Another simplification we employ is to model timers as simple integers, ignoring real-world constraints on timer values; we hope to handle timers more efficiently in future work.

Upon detecting a conflict, traces generated from the Spin model reveal an example execution that includes the initial state of the environment, the input event that triggers the conflict, the rules that are subsequently exercised, and finally the field that is updated more than once. It is then up to the user to either rewrite their program to prevent the conflict, or decide the conflict is acceptable as it stands.

It is worth mentioning that the Spin model can be used to check any invariant, not just the presence of conflict. One can imagine a more sophisticated interface that allows the end user to check any property they desire, for example "the front door should never be unlocked when no one is home" or "the porch light should remain on for no more than 30 minutes."

Conflict Detection in General-purpose Programming Languages. The three ECA platforms we studied could be analyzed by this style of conflict detection as well, although the semantics of Smart Lights and Home Assistant, in which the evaluation order of rules in response to an event is meaningful, would result in larger state spaces to explore. Applying this technique to general-purpose language platforms like OpenHAB and SmartApps would be more challenging. Although Spin can be used with general-purpose languages, as in the Java Pathfinder tool [18], any such model would have to include features like threading and memory allocation, greatly increasing its complexity. Such tools typically require more programmer input (adding annotations, using a special runtime) and are unlikely to be used by casual end-user programmers.

6.2 Provenance

As users live with their automations, they will inevitably experience unwanted or surprising behavior, prompting questions like, "Why did my lights turn on last night at midnight?" Collections of rules are unstructured, so it may be difficult for users to know how to edit their program to fix the problem, especially if the program is large. Moreover, it is infeasible to tweak rules and rerun them to test the change, as we might do in traditional debugging. An ideal answer to this question would give the sequence of events—and the rules that the events triggered—that ultimately led to the lights turning on at midnight.

To produce this answer, we need to analyze the particular execution that led to the unexpected behavior. Thus we need to retain information about the events registered and actions taken as the system runs in a timestamped log.

Execution traces. We augment the small-step semantics to produce a trace that marks each event, the rules triggered by the event, and a timestamp of when the event occurred.

$$(f[v_1 \hookrightarrow v_2], \{(h_1; p_1; a_1), (h_2; p_2; a_2), \cdots\}, \tau)$$

Note that the set of rules executed in response to an event may be empty. In order to generate traces, we augment three small-step rules: R-E marks and timestamps the event selected, and R-A records when an action is triggered in response to the event. Finally, R-AE marks generated events to distinguish them from events from the world. The semantics enforces a total order on events during evaluation, guaranteeing unique timestamps.

We initialize the trace by recording "world" events for every field in the environment that holds a value, using a dummy value for the old value and the field's initial value as its new value. These artificial events serve to record the state of the initial environment and do not trigger any rules.

Positive Queries. In a positive query, the user asks a question of the form "Why did field f have value v at time τ?" For example, the user might ask, "Why were the kitchen lights on at 3am last night?" We answer these questions by performing a dynamic program slice on the execution trace to find the rules and events that explain the field's value at the specified time.

In our dynamic slicing procedure, the slicing criterion is some environment field f and a point in the rule set's execution trace. Using this slicing criterion we construct a program dependency graph. Each entry in the trace, which are either world events or rule executions, is a node in this graph. Edges are placed between nodes if there is a data dependency or a control dependency between them. A rule execution has a data dependency with every trace entry that set the value of a field used in its predicate. Control dependencies link an event and a rule execution whose event handler was triggered by that event. Rule executions triggered by a timer have two control dependencies: the timer reaching some time, and the rule execution that set the timer.

Note that a rule execution will have one or two outgoing control dependency edges, and for every field referenced in its predicate, a data dependency edge. World event entries have no outgoing edges. We can thus determine how many outgoing edges each entry should have syntactically.

Positive Query Dynamic Slicing Algorithm. At a high level, we find all trace entries that belong to the program slice by marking the trace entry in which the field we are concerned with is set to the value it has at the control point in the slicing criterion. We then visit that trace entry by marking all nodes to which it has outgoing edges in the program dependency graph. We continue visiting and marking

[18] http://babelfish.arc.nasa.gov/trac/jpf, accessed 2016-11-15.

```
21:01:00 event PORCHMOTIONDETECTOR.sensor from
              NOMOTION to MOTION
     rule 3 [PORCHLIGHT.switch := ON, start
              timer.porchlight at 0]
 21:02:00 event FRONTDOOR.lock from locked to
                 unlocked
     rule 1 [HALLWAYLIGHT.switch := ON,
            BEDROOMLIGHT.switch := ON]
 21:03:00 event timer.porchlight from 0 to 1
21:03:38 event FRONTDOOR.lock from unlocked to
                  locked
 21:04:00 event timer.porchlight from 1 to 2
 21:05:00 event timer.porchlight from 2 to 3
21:06:15 event BEDROOMLIGHT.brightness from 75
                  to 25
     rule 2 [BEDROOMLIGHT.color := #0000ff]
   21:07 event timer.porchlight from 3 to 4
     rule 4 [PORCHLIGHT.switch := OFF, stop
              timer.porchlight]
 22:05:05 event BEDROOMLIGHT.switch from ON to
                  OFF
```

Figure 8. A sample trace; highlighted trace entries are part of the slice for the positive provenance query asking why the porch light was off at 22:00.

nodes until all marked nodes have been visited; the marked and visited nodes then make up the program slice.

However, since the trace is ordered by timestamp, there is no need to explicitly construct the program dependency graph. Instead, for each marked node, we can traverse the trace backwards, determining if each trace entry we encounter should be connected by an edge using the definitions above, stopping our traversal when all required edges have been found (or until we reach the beginning of the trace).

In Figure 8 we show an example execution trace. The provenance query for the porch light's state (off) at 22:00 returns the highlighted lines in the trace, showing the actions of the relevant rules (when the porch motion detector senses movement, the porch light is turned on and a timer is set; when the timer reaches 5 minutes, the porch light is turned off) and the events that triggered them.

Negative Queries. Users may also enquire about the *absence* of events: a user might ask "Why weren't the shades closed at 6pm today?" Intuitively, it is likely to be harder to explain why something didn't happen at a given time than why it did. We answer these questions by means of two mutually recursive algorithms: a negative state query (why didn't some field f have some value v at time τ), and a negative event query (why didn't some event e occur at time τ).

From the initial question "Why didn't field f have value v at time τ?", we find all rules that could set the field to that value. For each such rule we investigate why the rule did not set the field to the desired value at the specified time. First, we review the trace to find if the rule ever executed. If it did, we find the subsequent event that overwrote that value, and call the positive provenance query to find why the field had that value at that time. If it did not, there are two possible scenarios: either the event that matches the rule's event handler never occurred, or the event did occur, but at the time it occurred the rule's predicate evaluated to false. In either case, we call another recursive algorithm: if some event did not occur, we look for a reason why, in a manner analogous to the negative state query. If the event occurred but the rule's predicate was not true at that time, we can call positive or negative state queries to find why the fields referenced in the predicate did or did not hold the values that would make the predicate evaluate to true.

We have implemented both positive and negative provenance analysis to run on the execution traces produced by our Iota simulator, called via command line. Building a friendlier user interface and conducting a user study to determine if the analysis helps users find bugs is future work.

Provenance in General-purpose Languages. As with conflict detection, our provenance technique could apply to IFTTT, Smart Lights, or Home Assistant, assuming an execution trace is available. Since provenance operates on a known execution order, the difference in semantics between Iota and those platforms is less important. However, performing why/why not analysis on general-purpose programming languages is much more complex. Whyline's Java analysis required instrumenting Java byte code and handling threads, I/O events, and the state of the Java stack and heap. Furthermore, because Smart Apps are developed in isolation, the platform has no support for analyzing the behavior of a system of apps as a whole, even though apps may interact with each other's execution.

7 Related Work

Event-condition-action Languages. The databases community first proposed event-condition-action rules as a mechanism for efficiently responding to new data in databases attached to sources of streaming data [5]. However, these *active database systems* either had prioritized rules or a semantics that allowed for interleaved actions [16]. Iota proposes a semantics that avoids the former while still offering predictable behavior.

Trigger-action Programming. The ubiquitous computing community has proposed *trigger-action* programming as an end-user facing language for home automation [20]. Iota generalizes trigger-action programming and offers a precise semantics inspired in part by lessons from ubiquitous computing, such as avoiding the undesirable ambiguity that arises from mixing state and trigger events [7].

Guarded atomic actions [6, 17] are condition-action rules used in hardware synthesis. These rules do not have an event in the sense described in this paper. The main focus of the literature is on efficient compilation to hardware.

Conflict Detection. Several approaches for detecting conflicts in home automation programs have been proposed [14, 18]. Most define conflicts more broadly as feature interaction, which encompasses conflicting interactions on the environment (for example, turning a heater and a fan on at the same time). The work of Maternaghan and Turner [11] is similar to ours, but extends home automation programs with a policy definition tool that requires users to model interactions. Their work acknowledges that many conflicts will not actually represent problems and may be ignored by the user, as ours does. Zave *et al.* [22] propose conflict detection but resolve conflicts dynamically, by annotating rules with priorities. Leelaprute *et al.* [9] detect conflicts using the Spin model checker but adopt a different semantics than Iota does, considering interactions that result from simultaneous but independent events to be conflicts.

Provenance. Program slicing is a well-developed technique (see [19] for a survey.) Our provenance tool is most similar to Whyline [8], which addresses Java programs; a similar tool by the same authors, Crystal, allows users to ask why and why not questions about the behavior of user interfaces [12]. The tool Pervasive Crystal [21] applied this technique to context-aware applications such as museum displays, relying on programmer annotations to generate explanations. A study by Lim *et al.* [10] showed that why and why not explanations helped users reason about the behavior of context-aware systems such as a smart thermostat.

Other IoT Analyses. Several other analyses have recently emerged targeting a variety of home automation systems. Croft *et al.* detect temporal inconsistencies in HomeOS programs by reduction to timed automata [4], and Nandi *et al.* propose a static analysis for detecting and synthesizing missing event triggers in OpenHAB rules [15]. We hope to explore Iota support for these analyses as future work.

8 Conclusion and Future Work

We present Iota, a core calculus for Internet of Things Automation; use it as a basis to develop conflict detection and provenance in home automation programs; and compare Iota with existing platforms and show its expressivity. Iota occupies a "sweet spot," expressive enough to encode interesting home automation applications while remaining amenable to analysis.

Now that a precise semantics has been defined, we hope to use Iota to build more sophisticated languages or interfaces to better serve end users who are not expert programmers. For example, Iota could easily be lifted to a GUI or a natural language interface that could be either textual or voice-controlled. A language with a higher level of abstraction could be designed, adding features such as arithmetic, string manipulation, and variable assignment. A higher level of program structure should be explored beyond the bag-of-rules model discussed here; for example, should all rules with event handlers in common be composed together? A user study would be needed to gauge how high-level language features help users write, understand, and maintain automations.

In addition, we hope to add simulation, termination checking, and invariant specification and checking to our suite of analyses, and to explore type systems for Iota. Finally, we are eager to continue exploring the theoretical underpinnings of home automation, including a more precise relationship with timed automata [1].

A Appendix

A.1 Desugaring Groups

$$\frac{}{ds \vdash \texttt{devices} \rightsquigarrow \texttt{zip}\ ds\ \texttt{true}}$$

$$\frac{ads = \texttt{zip}\ ds \cap \{d_1, \ldots, d_n\}\ \texttt{true}}{ds \vdash \{d_1, \ldots, d_n\} \rightsquigarrow ads}$$

A.2 Desugaring Event Handlers

$$\frac{ds \vdash g \rightsquigarrow ads_1 \qquad ads_2 = \{(d, n \land p_b) \mid (d, p_b) \in ads_1 \land ds \vdash p_a[d/x] \rightsquigarrow n\}}{ds \vdash (g \mid x \to p_a) \rightsquigarrow ads_2}$$

$$\frac{ds \vdash g \rightsquigarrow ads \qquad c_1 = (d, p_a) \in ads \qquad c_2 = ds \vdash h[d/x] \rightsquigarrow ahs' \qquad c_3 = (h', p'_a) \in ahs' \qquad ahs = \{(h', p_a \land p'_a) \mid c_1 \land c_2 \land c_3\}}{ds \vdash \texttt{any}\ g\ (x \to h) \rightsquigarrow ahs}$$

A.3 Desugaring Predicates

$$\frac{ds \vdash g \rightsquigarrow ads \qquad ps = \{\neg p_d \lor p_b \mid (d, p_d) \in ads \land ds \vdash p_a[x/d] \rightsquigarrow p_b\}}{ds \vdash \texttt{all}\ g\ (x \to p_a) \rightsquigarrow \bigwedge ps}\ \textsc{d-all}$$

$$\frac{ds \vdash g \rightsquigarrow ads \qquad ps = \{\neg p_d \lor p_b \mid (d, p_d) \in ads \land ds \vdash p_a[x/d] \rightsquigarrow p_b\}}{ds \vdash \texttt{exists}\ g\ (x \to p_a) \rightsquigarrow \bigvee ps}\ \textsc{d-exists}$$

$$\frac{d \in ds \qquad d.f\ \text{exists}}{ds \vdash f \in d \rightsquigarrow \texttt{true}}\ \textsc{d-in}$$

$$\frac{d \notin ds\ \text{or}\ d.f\ \text{does not exist}}{ds \vdash f \in d \rightsquigarrow \texttt{false}}\ \textsc{d-notin}$$

A.4 Desugaring Actions

$$\frac{ds \vdash g \rightsquigarrow ads \qquad acs = \{(a[d/x], p) \mid (d, p) \in ads\}}{ds \vdash \mathtt{map}\ g\ (x \rightarrow a) \rightsquigarrow acs}\ \text{D-MAP}$$

A.5 Desugaring Rules

$$\frac{}{ds \vdash f[\cdot \hookrightarrow n]; p; as \rightsquigarrow f[\cdot \hookrightarrow]; f = n \wedge p; as}\ \text{D-TO}$$

$$\frac{}{ds \vdash f[n_1 \hookrightarrow n_2]; p; as \rightsquigarrow f[n_1 \hookrightarrow]; f = n_2 \wedge p; as}\ \text{D-FROMTO}$$

$$\frac{\begin{array}{c} ds \vdash h \rightsquigarrow ahs \qquad ds \vdash p_a \rightsquigarrow p_b \qquad ds \vdash as \rightsquigarrow acs \\ rs = \{h'; p_h \wedge p_b \wedge p_c; a \mid (h', p_h) \in ahs, (a', p_c) \in acs\} \end{array}}{ds \vdash h; p; as \rightsquigarrow rs}\ \text{D-RULE}$$

References

[1] Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theor. Comput. Sci.* 126, 2 (April 1994), 183–235. DOI: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*. Springer-Verlag, London, UK, UK, 193–207. http://dl.acm.org/citation.cfm?id=646483.691738

[3] AJ 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*. ACM, 2115–2124.

[4] Jason Croft, Ratul Mahajan, Matthew Caesar, and Madan Musuvathi. 2015. Systematically Exploring the Behavior of Control Programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 165–176. http://dl.acm.org/citation.cfm?id=2813767.2813780

[5] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. 1988. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Rec.* 17, 1 (March 1988), 51–70. DOI: http://dx.doi.org/10.1145/44203.44208

[6] James C. Hoe and Arvind. 2000. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000*. 511–518. DOI: http://dx.doi.org/10.1109/ICCAD.2000.896524

[7] 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

[8] Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, New York, NY, USA, 151–158. DOI: http://dx.doi.org/10.1145/985692.985712

[9] Pattara Leelaprute, Takafumi Matsuo, Tatsuhiro Tsuchiya, and Tohru Kikuno. 2008. Detecting feature interactions in home appliance networks. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD'08. Ninth ACIS International Conference on.* IEEE, 895–903.

[10] Brian Y Lim, Anind K Dey, and Daniel Avrahami. 2009. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2119–2128.

[11] Claire Maternaghan and Kenneth J Turner. 2013. Policy conflicts in home automation. *Computer Networks* 57, 12 (2013), 2429–2441.

[12] Brad A Myers, David A Weitzman, Andrew J Ko, and Duen H Chau. 2006. Answering why and why not questions in user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 397–406.

[13] Alessandro A. Nacci, Bharathan Balaji, Paola Spoletini, Rajesh Gupta, Donatella Sciuto, and Yuvraj Agarwal. 2015. BuildingRules: A Trigger-action Based System to Manage Complex Commercial Buildings. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers (UbiComp/ISWC'15 Adjunct)*. ACM, New York, NY, USA, 381–384. DOI: http://dx.doi.org/10.1145/2800835.2800916

[14] Masahide Nakamura, Hiroshi Igaki, and Ken-ichi Matsumoto. 2005. Feature interactions in integrated services of networked home appliances. In *Proc. of Int'l. Conf. on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI'05)*. 236–251.

[15] Chandrakana Nandi and Michael D Ernst. 2016. Automatic Trigger Generation for Rule-based Smart Homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 97–102.

[16] Norman W. Paton and Oscar Díaz. 1999. Active Database Systems. *ACM Comput. Surv.* 31, 1 (March 1999), 63–103. DOI: http://dx.doi.org/10.1145/311531.311623

[17] D. L. Rosenband and Arvind. 2005. Hardware synthesis from guarded atomic actions with performance specifications. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005*. 784–791. DOI: http://dx.doi.org/10.1109/ICCAD.2005.1560170

[18] Yan Sun, Xukai Wang, Hong Luo, and Xiangyang Li. 2015. Conflict detection scheme based on formal rule model for smart building systems. *IEEE Transactions on Human-Machine Systems* 45, 2 (2015), 215–227.

[19] Frank Tip. 1995. A survey of program slicing techniques. *Journal of programming languages* 3, 3 (1995), 121–189.

[20] 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

[21] Jo Vermeulen, Geert Vanderhulst, Kris Luyten, and Karin Coninx. 2010. PervasiveCrystal: Asking and answering why and why not questions about pervasive computing applications. In *Intelligent Environments (IE), 2010 Sixth International Conference on.* IEEE, 271–276.

[22] Pamela Zave, Eric Cheung, and Svetlana Yarosh. 2015. Toward user-centric feature composition for the Internet of Things. *arXiv preprint arXiv:1510.06714* (2015).