Protecting Smart Homes from Unintended Application Actions

Aqsa Kashaf
Carnegie Mellon University
akashaf@andrew.cmu.edu

Vyas Sekar
Carnegie Mellon University
vsekar@andrew.cmu.edu

Yuvraj Agarwal
Carnegie Mellon University
yuvraj@cs.cmu.edu

Abstract

Many smart home frameworks use applications to automate devices in a smart home. When these applications interact in the same environment, they may cause unintended actions which can lead to a safety violation (e.g., the door is unlocked when the user is not at home). While recent efforts have attempted to address this problem, they do not capture complex app behaviors such as: 1) *timed behavior* and *user inputs* (e.g., a door can remain unlocked for a long time because of a lock-door app that locks the door after x duration, if x is set too large.) and 2) interactions between devices and the environment they implicitly affect (e.g., water sprinklers cannot be turned on if the water supply is off). Hence, prior work leads to many false positives and false negatives. In this paper, we present PSA, a practical framework to identify safety intent violations in a smart home. PSA uses parameterized timed automata (PTA) as an expressive abstraction to model smart apps. To parse these apps into PTA, we define mappings from smart app APIs to equivalent PTA primitives. We also provide toolkits to model devices, environments, and their interactions. We evaluate PSA on 86 apps in the Samsung SmartThings IoT ecosystem. We compare PSA against two state-of-the-art baselines and find: (a) 19 new intent violations and (b) 35% fewer false positives than baselines.

Keywords

IoT safety and security, violation detection, formal verification

1 Introduction

With the surge in IoT devices, several smart home platforms have emerged; e.g., Samsung SmartThings [34], IFTTT [20], which allow users to control devices using apps. Unfortunately, these apps can cause safety violations when they interact directly or because of shared environmental context. Apps can also be incorrectly configured by users, leading to unsafe situations; e.g., in Figure 1, if the user sets a large value of x, the door will remain unlocked for a long time. Recent work attempts to detect such violations in IoT ecosystems either statically [8, 9, 11, 23, 31, 37, 40] or dynamically[10, 38]. However, these efforts are not expressive enough to capture the following features of modern smart apps and IoT ecosystems:

1. *Timing:* Consider the interactions of real SmartThings apps in Figure 1. The *unlock-when-arrive* app unlocks the door when the user arrives. After the user enters and closes the door, *auto-lock-door* app is triggered, which waits for x min and then locks the door. If we do not model the *wait*, it is not a safety violation because the door is locked after unlocking. However, it is a safety violation if x is large. Existing efforts ([9, 11, 23, 37]) either ignore timing or only model it coarsely, resulting in inaccuracies.
2. *User Inputs:* Apps can be customized by users, e.g., the auto-lock-door app in Figure 1 takes input x. These inputs often determine if an app is safe. However, existing techniques ([8, 9, 23, 30, 31, 37, 40]) do not reason about the safe range of user inputs.

Figure 1: A violation that exists only if the value of x is large.

3. *State:* Apps can maintain state (discrete value or time), e.g., consider an app that turns off an A/C if it has already run for 2 hours. The app tracks when the A/C was run by keeping state. Unfortunately, prior works either consider stateless apps ([8, 23, 37, 40]) or capture discrete state only ([9, 11]), while failing to capture continuous state resulting in false positives or negatives.
4. *Environment Interaction:* Apps may interact indirectly because of shared environment and devices. For example, if an app turns the main water valve off when a leak is detected, another app cannot turn water sprinklers on. We need to model this interaction to detect that sprinklers cannot be run. Existing efforts ([8, 9, 11, 30, 31, 40]) either miss or only partially capture these interactions resulting in false negatives or false positives.

To overcome the limitations of prior work, we present PSA. [1] PSA uses model-checking to verify a smart home deployment for safety violations. It uses static instead of runtime analysis which allows PSA to detect violations *before* they occur. PSA takes as input a set of apps deployed in a home (with their source code) and a set of safety properties or *intents* which check if an app does anything unintended. It then confirms whether the apps satisfy these intents or provide counterexamples when they do not.

In designing PSA, we need an expressive abstraction to model the stateful and timed behavior of smart apps, and their interaction with the devices and the environment. Hence, we explore the abstractions proposed in the model-checking literature [3, 13, 21, 35]. We focus on literature in the cyber-physical systems (CPS) since CPS have an additional notion of environment interactions which differentiate it from verification in software systems. We identify *parameterized timed automata* (PTA) as a suitable abstraction. Next, it is hard to translate the apps correctly into PTA, as there is no one-to-one mapping between apps and PTA. We carefully define these app to PTA mappings and verify our generated PTA for correctness by emulating an app in SmartThings, and comparing the output trace with its PTA. We also define a taxonomy for environment attributes and sensor devices, and provide templates to model each category of devices and environment attributes into PTA in the context of smart home. Moreover, we define a mapping of interactions between device actions and environment attributes by building a device-environment library. We describe these in Section 5 and 6. In this paper, we make the following contributions:

- We identify PTA as the expressive and appropriate abstraction to model modern smart apps. We define a mapping between various smart app APIs into equivalent PTA primitives such that it is correct and efficient. Finally, we design and implement a parser to generate formal models of apps from source code.

[1] The name PSA stands for Proactive Safety for smart home Applications, which is also a play on Public Safety Announcements for Smart Homes.

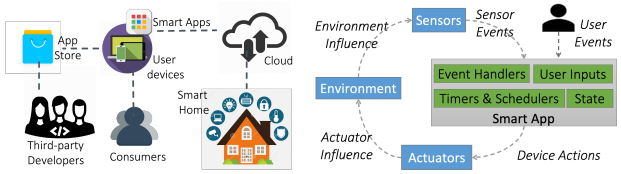


Figure 2: An overview of the Smart Home Ecosystem (a) and a logical workflow between different components of this smart home (b).

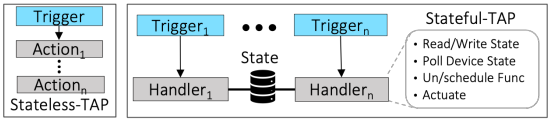


Figure 3: Trigger-action programming (Stateless, Stateful).

- We develop a systematic methodology to model devices, environment, and intents for smart homes. We design heuristics to make the model-checking more tractable. For instance, we discretize continuous inputs e.g., temperature. We model the *environment effects* qualitatively and not quantitatively and, use taint-tracking to minimize the number of variables while parsing the apps.
- We run PSA on 86 SmartThings apps and find 640 intent violations. We also find that 23% of these violations can be avoided when configured using the safe values output by PSA.
- We show that PSA has no false positives or false negatives (when our analysis terminates). We also compare our system against two baseline modeling approaches [9, 23], [2] and show that PSA leads to 35% less false positives and finds 19 new violations.

2 Background and Motivation

We begin with a background on modern smart home ecosystems. Then, we give a taxonomy of *safety intent violations* and discuss the limitations of prior work.

2.1 Smart Home Ecosystem

Smart home platforms, such as SmartThings [34] (Figure 2a), usually comprise a cloud-connected hub to control the IoT devices. Users interact with IoT devices in their home using *smart apps*, from the provider's [33] or third-party app store [7]. Figure 2b shows a logical view of a smart home. Changes in the environment influence the sensor devices (e.g., temperature influences thermostat reading). The sensor devices trigger apps, which actuate on devices, affecting the environment (e.g., A/C.on() to decrease temperature).

Smart Apps: Smart apps have event-driven programming, also called trigger-action programming (TAP). There are two types of TAP paradigms for smart apps (Figure 3): 1) Stateless-TAP apps (e.g., IFTTT [20], Zapier [39]), which have stateless event handlers (e.g., turn lights on when there is motion), 2) Stateful-TAP apps (e.g., SmartThings [34], Hubitat [19]), which preserve the context across executions, such as counting the frequency of an event. These smart apps can also schedule actions and keep timers (e.g., turn A/C off after 30min). Stateful-TAP is a super-set of Stateless-TAP apps. In this work, we consider Stateful-TAP apps, focusing on Samsung SmartThings [34] since it is widely used. However, our techniques also apply to other platforms (e.g., IFTTT [20], Hubitat [19]).

[2]These baselines are approximations of prior work.

```
1  preferences {
2   input "AC", "capability.switch",  input "threshold", "Number"
3   input "TH", "capability.thermostat",  input "d", "time"}
4  def installed() {
5   subscribe(TH, "temperature", handler) }
6  def handler(evt) {
7   if (evt.value < threshold) {
8    AC.off()
9    runIn(60 * d, func)}
10 def func() { AC.on()  }
```

Figure 4: Example SmartThings application which turns A/C off for duration d when temperature is below a threshold.

Smart apps are also configurable; users can set app parameters e.g. temperature threshold x for switching on A/C. Figure 4 shows an example smart app that turns A/C off for `duration` minutes when the temperature drops below a threshold. The `preferences` section (line 1-5) specifies inputs which are configured by the user at install time. In this example, the user needs to select an A/C, a thermostat, a temperature threshold, and the duration. The app subscribes to events in the `installed` section (line 6-7) at the install time. Each subscribed event has a handler which is triggered when the event takes place. This example also highlights the timed behavior in apps. We discuss the detailed features of smart apps in Section 5.

2.2 Safety Violations in Smart Home

We first present a taxonomy of intent violations that we use throughout the paper. It captures all the safety violations proposed in the prior work [9, 23, 31, 37], as well as novel extensions to incorporate the effects of timed behavior. To define the violations, we introduce some notation. Let $a_{d,1}$,$a_{d,2}$,...,$a_{d,n}$ be the list of actions that can be performed on device d. For example, for an air-conditioner (A/C) the actions can be A/C.on() or A/C.off(). Let $t_{a_{d,1}}$ be the time when action $a_{d,1}$ on device d happened. Let $condition(s_d)$ be a boolean operation defined on device state s_d of device d e.g., a condition defined on the A/C state can be: *if A/C is on*. Let e^k_m be the mth state of environment attribute e^k e.g., the A/C.on() action affects the environment attribute "temperature".

P1 Conflicting Actions: Consider the following two interactions:

$$thermostat\_reading > 75F \rightarrow windows.open()$$
$$presence\_sensor.absent \rightarrow windows.close()$$

Here, the two interactions lead to conflicting actions. Conflicting actions can be of two types:

P1.1 Device Conflicts: Conflicting actions happening on the same device are device conflicts. Formally, the following interactions between sensor events and actuator actions result in device conflicts:

$$condition(s_{d_1}) \rightarrow a_{d_2,1}\ (1) \qquad condition(s_{d_3}) \rightarrow a_{d_2,2}\ (2)$$
$$where\ a_{d_2,1} \neq a_{d_2,2}\ and\ |t_{a_{d_2,1}} - t_{a_{d_2,2}}| \leq x\ ; x\ is\ some\ duration$$

P1.2 Environment Conflicts: These occur when app actions cause conflicting effects on the same environment attribute. Formally:

$$condition(s_{d_1}) \rightarrow a_{d_2,} \rightarrow e^k_1\ (1) \qquad condition(s_{d_3}) \rightarrow a_{d_4,} \rightarrow e^k_2\ (2)$$
$$where\ e^k_1 \neq e^k_2,\ d_2 \neq d_4\ and\ |t_{a_{d_2,}} - t_{a_{d_4,}}| \leq x\ ; x\ is\ some\ duration$$

For example, one app turns on a heater causing the temperature to increase and another app turns on an A/C causing it to decrease.

P2 Co-occurrence Violation: Certain device states and actions should always co-occur. For example, when no one is in the house,

	User Config	State	Env. Model.	Time Model.	Intent Vio.	Prog. Paradigm
SiFT [23]	✗	✗	✓*	event	P1,2,4	Stateless
Soteria [9]	✗	discrete	✗	✗	P1.1,2	Stateful
IoTSan [31]	✗	✓	✗	✓	P1.1,2	Stateful
Iota [30]	✗	✓	✗	✓	P1.1,2	Stateful
HomeGuard [11]	✗	discrete	✗	✗	P1.1,2	Stateful
iRuler [37]	✗	✗	✓	event	P1,2,4	Stateless
AutoTap [40]	✗	✗	✗	✓	P1.1,2,3	Stateless
Menshen [8]	✗	✗	✓	✓	P1,2	Stateless
Salus [22]	✓	✗	✓	event	P2	Stateless
PSA	✓	✓	✓	✓	P1-5	Stateful

Table 1: Comparison of PSA with prior work (Vio means violation)

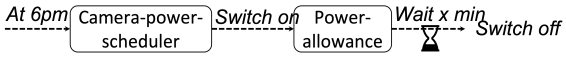


Figure 5: Example of a device conflict that we found. It exists only if the value of x is small

the door should be locked, $presence\_sensor.inactive \rightarrow door.lock()$. Formally, in $s_{d_1} \rightarrow a_{d_2,}$, $a_{d_2,}$ should occur with device state s_{d_1}. In $e_1^k \rightarrow a_{d_1,}$, $a_{d_1,}$ and environment attribute state e_1^k should co-occur. This property checks if the co-occurrence is violated. There can be a conjunction of multiple device or environment states.

P3 Deadline Violation: Certain actions should be time bound for user safety. For example, the door should be locked within a min of being closed ($door.unlocked() \rightarrow door.close()$ within 1 min). If the action is not performed within a given deadline, then it is a *deadline* violation. These deadlines are specified on device actions. More formally, $a_{d_1,} \rightarrow eventually\ a_{d_2,} within\ x\ duration$. We can also have a conjunction of actions on either side.

P4 Blocked Action: Consider the following interaction:

$$leak.detected \rightarrow water\_supply.off()$$
$$fire.detected \rightarrow fire\_sprinkler.on()$$

Here the action $water\_supply.off()$ blocks the action $sprinkler.on()$. If an action $a_{d_1,}$ on device d_1, makes it impossible to execute another action $a_{d_2,}$ on device d_2, then $a_{d_2,}$ is a blocked action. More formally, $a_{d,} \rightarrow never\ a_{d'}$, Here d_1 and d_2 can be the same device.

P5 Invalid Action Sequence: This is a general property to capture unreasonable, or risky action sequences; e.g., if the duration between $door.close()$ and $door.lock()$ action is over 3 min, then it is a risky action sequence. More formally, invalid action sequence between two actions can be of the form:

$$(a_{d,}, t_{a_{d,}}), ..., (a_{d'}, t_{a_{d',}})\ where\ |t_{a_{d',}} - t_{a_{d,}}| \diamond x,\ \ \diamond = \{=, <, >, \leq, \geq, \neq\}$$

x is some duration. Note that P1 and P3 are special cases of P5.

2.3 Comparison with Prior Work

Given this context of apps and violations, we now compare the existing approaches with our work by looking at the various aspects of smart home frameworks in Table 1. The columns 1-4 in Table 1 highlight the supported app features. The fifth column highlights the supported properties mentioned in Section 2.2. The last column reports the supported programming paradigm for each approach.

User configurable inputs: Apps may have configurable inputs which often decide whether a configuration is safe. For example, in Figure 1, the scenario is unsafe only if x is large. Existing techniques [8, 9, 11, 23, 30, 31, 37, 40] do not analyze the apps for potential safe configurations as they take pre-configured apps. Salus [22] gives some insight into safe configurations for the stateless-TAP apps. But, it cannot be extended to the stateful-TAP apps.

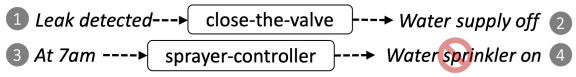


Figure 6: A blocked action violation that we found, where water sprinkler cannot turn on because main water supply is off.

Modeling time: Smart apps exhibit timed behavior (Section 2.1). Modeling time is important to check for intent violations accurately. For instance, consider the interaction between actual SmartThings apps in Figure 5. The camera-power-scheduler app turns on a switch for charging the camera and the power-allowance app turns it off after duration x. If we do not model the wait after the *switch on* action, we will incorrectly mark it as a device conflict, which exists only if the actions happen together, i.e., when x is small. Apps show timed behavior in two ways: 1) Timed Events which are triggered on the value of time during the day, e.g., the camera-power-scheduler app is triggered at 6 pm. 2) Duration, where some time interval is measured between events and actions, e.g., the power-allowance app waits for x min before turning off the switch.

SiFT [23], iRuler [37] and Salus [22] model timed events but not duration as they only consider stateless-TAP apps. Soteria [9] and HomeGuard [11] consider stateful-TAP apps but ignore their timed behavior. These approaches will mark the interaction in Figure 5 as a device conflict, even if x has a large value (i.e., a false positive), and also miss the violation in Figure 1 (i.e., a false negative). Also, properties having time e.g., P3 cannot be handled by previous work (e.g., [9, 11, 22, 23, 30, 37]). Other approaches [30, 31] model timed behavior in apps, but only support untimed properties.

State: Apps can maintain state, which may be a discrete value or time. For example, consider an app that tracks when the A/C was last run and turns it off if it has run for 2 hours. Prior work (SiFT [23], iRuler [37], AutoTap [40], Salus [22] and Menshen [8]) only consider stateless apps and cannot be trivially extended to handle stateful apps. Soteria [9] and HomeGuard [11] do not model time and hence can only handle discrete state.

Modeling Environment: Apps may have indirect inter-app interactions when they run in the same environment. For example, in Figure 6, when the water supply is shut, the sprinklers cannot be run, causing a blocked action (Property P4). We cannot detect this blocked action if we do not model the interaction between the water supply and the sprinklers. To find environment related violations (e.g., P 1.2), we need to model the interactions between devices (sensors and actuators) and the environment. Existing techniques either do not model these interactions [9, 11, 30, 31, 40] or only model interactions between actuators and the environment [8, 22, 23, 37].

Programming Paradigm: Smart apps follow two programming paradigms, Stateless- and Stateful-TAP (Section 2.1). The latter is more expressive and enables complex automation e.g., Figure 4. Most of the prior works ([8, 22, 23, 37, 40] support stateless-TAP, which cannot be trivially extended to stateful-TAP apps.

3 System Overview

In this section, we describe PSA (Figure 7) and highlight our design challenges. PSA uses a model checking approach [12] to verify whether a smart home deployment (set of apps deployed) satisfies the specified safety intents. We envision PSA to be used offline by

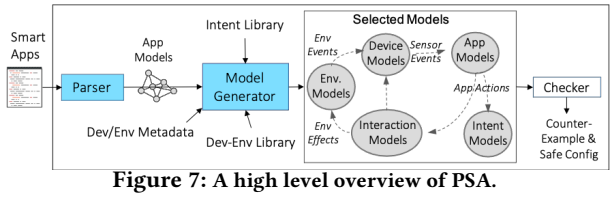


Figure 7: A high level overview of PSA.

the smart home vendor (e.g. SmartThings). The vendor will run the analysis for all possible deployments (set of apps in a house).

PSA needs the source code of the apps like prior work [9, 11, 23, 30, 31]. Users can specify a set of device types as input to PSA. Otherwise, PSA checks for all available device types. Device type is a general category of devices such as A/C, heater, and not a specific instance. PSA has a built-in library of safety intents (Section 2.2). PSA outputs a counter-example in case of a violation, and a set of safe configurations of user inputs for the apps. The smart home vendor can then provide this list of safe configurations to the users for their home deployments. The main components of PSA are:

- The *parser* converts an app into a formal model. In particular, for an app A_i of a smart home platform, a platform-specific parser takes the source code of A_i as input. It then generates an ensemble of models $M_1^i, M_2^i, ..., M_n^i$. The number of models depends on the number of event handlers, and asynchronous APIs used in the app, as explained in Section 5.2.
- The *model generator* creates models for the devices and the environment by looking at the app models, the safety intents, and the metadata for the devices and the environment attributes. It also creates interaction models to map device actions to the environment, using the device-environment library (Section 6) e.g., $A/C.on() \rightarrow temperature.decreasing$. For example, consider the violation in Figure 6. The model generator creates the device model for the *water sensor*. The environment model will be for the environment attribute *water* with states on and off. The interaction models will map the actions $water\_supply.off()$ and $sprinkler.on()$ to the model for water. This interaction is present in the device-environment library. Finally, the intent P4 checks if $sprinkler.on()$ is issued after $water\_supply.off()$.
- The generated models are then fed to a *model checker*. In case of a violation, the model checker outputs a counter-example as a trace of dated events and actions. PSA also outputs a set of configurations of input values that do not lead to an intent violation. This set can aid users to safely configure the apps.

Given this overview, there are a few outstanding questions:

Modeling Abstraction: As motivated earlier (Section 2), we need to model user inputs, timed behavior of apps, and their interaction with the environment to detect violations accurately. Modeling these entails trade-offs between the expressiveness of models, accuracy of verification, and performance. For example, we can model time quantitatively or qualitatively. Similarly, user inputs can be modeled by enumerating all values or by treating them as symbolic states. We discuss our choices in Section 4.

App to Model Parsing: Naively translating the apps to our abstraction can affect the expressiveness and correctness of PSA. For example, apps have synchronous and asynchronous functions, local and global variables, built-in utilities, e.g., find, collect, etc., diverse data types such as maps, lists, etc., scheduling and unscheduling of actions. We discuss these challenges and our approach in Section 5.

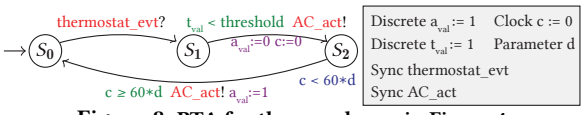


Figure 8: PTA for the app shown in Figure 4

Practical Smart Home Modeling: Besides apps, modeling the environment and devices is also challenging. For example, continuous environment attributes such as temperature, humidity, can significantly blow up the state space of models. Moreover, modeling the effects of various device actions on the environment is challenging as different devices can affect an environment attribute differently. For example, an A/C.on() and a fan.on() action, both affect temperature differently. We discuss this in Section 6.

4 Modeling Abstraction

Now we explain our choice of parameterized timed automata (PTA) as the modeling abstraction to model the smart home components such as smart apps, devices, environment, and intents.

Choosing the modeling abstraction: Our abstraction should allow us to model modern smart app features such as state, time, and user inputs, and other components of the smart home, such as devices. We looked at modeling options across three dimensions: 1) Time, 2) Environment Attributes, and 3) User inputs (Appendix A). We can model quantitative time either as discrete or continuous. Modeling time as discrete values would enable using a simple FSM abstraction. However, to achieve correctness we need to use sufficiently small intervals for discretization, leading to a large state space. On the other hand, modeling time as a continuous variable using either a Timed Automata [4], Timed Petri-Nets [32] or using Hybrid Automata [18] addresses this shortcoming.

While environment attributes are naturally continuous variables, we observe that they generally have a narrow discrete range of values in practice. For example, thermostat set-points are usually set at integer values within a narrow range (e.g. 60F-80F), sound and motion sensors usually produce a binary output indicating presence, and carbon-monoxide (CO) sensors need to check CO levels for particular thresholds. Therefore, we model environment attributes using discrete instead of a continuous variable. This saves us from state space explosion without compromising on accuracy.

We can model user inputs by either enumerating over all values (*enumeration*), or treating user inputs as symbolic variables/parameters (*parameterization*). We adopt parameterization since an app may have more than one user input leading to exponential complexity in the possible combinations with enumeration.

The above modeling choices lead us to choose Parameterized Timed Automata (PTA), where we model time as a continuous variable and environment attributes as discrete, using the parameterized variant of Timed Automata (PTA) to handle user inputs. Now we give a brief background on parameterized timed automata (PTA) using the example of the app mentioned in Figure 4.

Parameterized Timed Automata: A PTA extends a non-deterministic FSM with a finite set of real-valued *clock* variables, integer-valued *discrete* variables and unknown variables called *parameters*. A transition can be synchronized with other PTAs using synchronization labels (sync). Figure 8 shows the PTA model for the app in Figure 4 with states (S_0, S_1 and S_2), a clock c, discrete variables a_{val} and t_{val}, a parameter d, and sync $thermostat\_evt$ and AC_act.

A linear constraint over clocks, discrete variables, and parameters may *guard* a transition i.e., a transition can be taken only if the guard is satisfied. For example, $t_{val} < threshold$ guards the transition from S_1 to S_2. Clocks and discrete variables can be updated along with a transition. A clock can be updated to a linear combination over clocks, parameters, and discrete variables; a discrete variable can be updated to a linear combination over discrete variables. For example, from S_1 to S_2, c is reset to 0 and a_{val} is updated to 0. A clock tracks the elapsed time since its last reset and all the clocks evolve at the same rate. A state in a PTA may have an *invariant* as a linear constraint over clocks, discrete variables, and parameters e.g., the state S_2 has the invariant $c < 60 * d$. It means that the PTA can stay at S_2 for no longer than $60 * d$ time units.

To summarize, the PTA in Figure 8 starts at S_0 and transitions to S_1 when the sync *thermostat_evt* is received from the thermostat model (not shown here). We use ? for a receiving sync (e.g. *thermostat_evt*), and ! for a sending sync (e.g., *AC_act*) . At S_1, if the guard is satisfied, it synchronizes with the A/C model on sync *AC_act* and updates a_{val} to 0 for an *AC.off()* action, and resets the clock c. The PTA waits for time $60 * d$ at S_2, and then returning to S_0 and updates a_{val} to 1, while synchronizing with an A/C model using sync *AC_act*. Multiple PTAs are usually needed to model a system. Clocks, discrete variables, and parameters are shared among all PTAs. Transitions with the same sync across PTAs can be taken only if all the PTAs containing this sync can take the transition.

5 Modeling Smart Apps

As discussed in Section 3, naive parsing of apps into PTAs can cause scalability, expressiveness, or correctness issues. In this section, we first elaborate the main app features and then discuss the modeling choices we considered and their trade-offs. Finally, we explain our parsing algorithm to auto-generate PTA models from source code.

5.1 Features of Modern Smart Apps

We discuss some of the app features in Section 2.1. Table 2 shows a more exhaustive list of features. SmartThings apps inherit basic features (e.g., values (v), expressions (e) and control flow statements) from its underlying programming language Groovy. Besides, these apps also support domain-specific constructs.

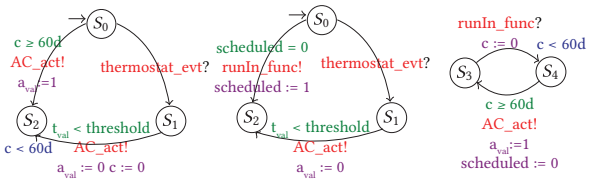
First, a smart app can store and access values in a database using the `state.x`, construct where `x` is a user-defined entry in the database. For example, an app that counts the number of events can use `state.counter` which is incremented when an event is received. Apps use the state construct to store *discrete* or *time* values.

Second, a smart app can interact with devices by receiving events or sending actions to them. It can also poll for the current state of a device using `currentState`, e.g., reading the temperature value from a thermostat `TH` using `TH.currentState("temperature")`. In addition, a smart app can get all events issued from a device `dvc` from a specific time `t` using `dvc.eventsSince(t)`, called *device history polling*.

Finally, smart apps offer an extensive set of constructs for timed behavior. For example, an app can schedule actions using `runIn(e,f)` (which calls function `f` at a time determined by the expression `e`) or do things periodically using `runEvery(e,f)` (which repeatedly runs `f` after some duration determined by `e`), `schedule(e,f)` (which runs `f` everyday at the time determined by `e`). Similarly, an app can use `unschedule(f)` to unschedule a previously scheduled execution of `f`.

Feature	Example constructs
Basics	values, expressions (e.g., `a+b`), `if`, `while`, `s1;s2`
Global state	`state.x`
Device interaction	`event`, `action`, `currentState`, `eventsSince`
Timed commands	`now`, `runIn`, `runEvery`, `schedule`, `unschedule`

Table 2: Constructs of SmartThings apps



(a) Blocking **(b) Non-blocking**

Figure 9: Two ways to model a `runIn` command

5.2 Modeling Challenges and Approach

Given the aforementioned app features, we discuss their modeling challenges and our approach to addressing them.

Timed Commands: We can treat timed commands like regular synchronous function calls. Figure 9a shows the model of the app in Fig 4 using this approach. The PTA waits at S_2 for $60 * d$ duration before executing the action $AC.on()$ and returns to the initial state. The PTA is blocked at state S_2 for $60 * d$. But, the timed commands in Table 2 are *non-blocking*. They schedule a function and return. A scheduler in the smart home framework executes these functions at the specified time. While a function is waiting to be executed, another execution of the app may *unschedule* the function or *update any state* which the scheduled function uses. Hence, we need to model the timed commands in a non-blocking way.

A naive solution is to add a self-transition at S_2 in Figure 9a that receives *thermostat_evt*. But we still cannot update state variables or unschedule calls. Another solution is to add a *True* transition from S_2 to S_0, introducing non-determinism. Now a later app execution can happen and update state and unschedule actions. But, if that transition is taken, the scheduled action will not happen. Hence, we create a separate PTA for each timed command shown in Figure 9b. The PTA synchronizes with another PTA on *runIn_func* sync at S_2. The other PTA waits for time $60d$ at S_3 and executes the action, while the first PTA returns to S_0. When a function is scheduled multiple times by subsequent calls, PSA only model the execution of the latest scheduled function, which is also done by SmartThings. Note that modeling the earliest one is straightforward.

Device Interactions: To model app interactions with devices, we use the PTA primitive *sync*, that allows apps models to synchronize with device models. For example, in Figure 9b, the sync *thermostat_evt* synchronizes transition between thermostat model and the app model. Besides events and actions, smart apps also poll the current and past device states. We implement current device state polling as reading the device value from its model. We discuss polling in Section 6.2. Note that PSA can also model event polling as compared to trigger-action. We do not model polling device history.

Global State: We model discrete-valued state using discrete variables e.g., a *state.counter* variable that counts the total events received. State variables are shared among the functions in an app, while local variables are local to a particular function. PSA models both using discrete variables. We append function identifiers to local variables to differentiate them across functions.

```
1  subscribe(motionSensor,"motion.active",m_handler)
2  subscribe(contactSensor,"contact.open",c_handler)
3  def m_handler(evt) {
4      state.motion = now()
5  }
6  def c_handler(evt) {
7      if(now() - state.motion > 3) {
8      //do something
9  }
```

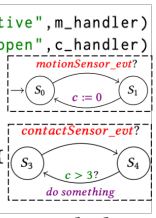
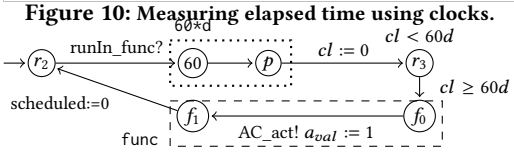
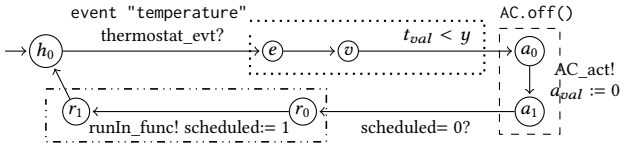


Figure 10: Measuring elapsed time using clocks.



(a) PTA for `runIn`



(b) PTA for `handler`

Figure 11: Illustration of parsing the example app for Figure 4

State variables also store time. Figure 10 shows a code snippet that stores time, when the *motion.active* event is received. On receiving the *contact.open* event, it measures the elapsed time between the two events. We can naively model it as we model discrete state, and replace discrete variables with clocks. Hence, there will be a clock c_1 for state.motion, and another clock c_2 for the call to now(). At line 4, the PTA will assign the value of c_2 to c_1 as $c_2 = c_1$. Then c_1 will be stopped at all states so that its value does not change. At line 7, the PTA will do $c_2 - c_1$. But, having stop-watches in model significantly increases verification time. Essentially, time-valued state variables measure the elapsed time. Recall that in PTA, clocks also measure the elapsed time since their last reset. Hence, we introduce a clock c for *state.motion*, and reset it when we receive *motion.active* at S_0 in Figure 10. When *contact.open* event is received, the value of c gives the elapsed time since *motion.active*. We parse line 7 as $c > 3$. This approach results in less number of clocks and no stop-watches.

Unschedule: An `unschedule(f)` call requires removing function `f` from the list of scheduled functions. PSA uses a flag variable to track if unschedule has been called on a function `f`. If the flag is true, the transition to the PTA of `f` is not taken.

We also provide primitives to capture other features such as values, expressions, and conditionals, omitted here for brevity.

Unsupported APIs: We do not model polling past device states (`eventsSince`) as there can be an indefinite number of past states.

5.3 Parsing Algorithm

Our parsing algorithm has two phases. In the first phase, we parse the `preferences` section in the app code to get the names and types of all user inputs. We also parse `subscribe` calls to generate a map from events to handlers. We use this map to generate PTA in a bottom-up fashion for each handler function in the second phase. We also do taint analysis to create our model with minimum variables.

Example: We illustrate the parsing of the app in Figure 4. First, consider the parsing of the `runIn` function (Figure 11a). The algorithm first parses the expression `60*d`. It generates a PTA (highlighted with dotted lines) with one state for 60 which is parsed as a constant and another state for d which is parsed as a $parameter(p)$. The return value of the PTA is $60d$. For function call `func`, which contains an `AC.on()` action, the generated PTA issues a sync AC_act and sets a_{val} to 1 for $AC.on()$ (highlighted in dashed lines). To finish the parsing for `runIn`, the two PTAs are composed as shown in Figure 11a. The algorithm adds a state r_3 with invariant $cl < 60d$ and adds a transition to $func$ at f_0 with guard $cl >= 60d$. To model `runIn` command (5.2), we also add a sync to synchronize with the handler PTA.

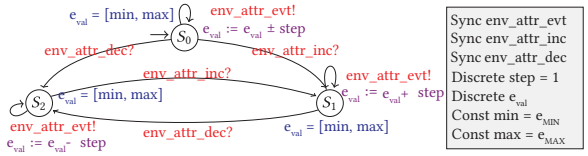


Figure 12: PTA Model for a non-categorical environment attribute.

Now, we consider the handler function, as shown in Figure 11b. First, the algorithm parses the condition and the statements appearing in the `if` statement. The parsing of the condition is highlighted with dotted lines, and the first statement in `if` (`AC.off()`) is highlighted with dashed lines. The second statement which is the `runIn` command is shown in the dotted-dashed line. We add $runIn_{f}unc$ to synchronize with the *runIn* PTA. The variable *scheduled* tracks if $func$ has been scheduled before. This example executes the earliest schedule. These PTA are later composed together to model the handler function. Finally, our algorithm adds an initial state h_0 for the handler, a transition from the final state r_1 of the PTA for the `if` statement, and a transition from h_0 to e (the initial state of `if`) with a sync thermostat_evt indicating the subscribed event.

6 Modeling Devices, Environment and Intents

In this section, we explain how we model the environment attributes, devices, and safety properties.

6.1 Modeling Environment

The environment models: 1) trigger changes in sensor states, and 2) react to app actions. We use the term *environment events* for triggers that cause a state change in sensors, and *environment effects* for changes in environment attributes that result from app actions on the devices e.g., a *heater.on()* causes the temperature to rise. Environment attributes can be categorical (e.g., motion can either be present or absent) or non-categorical (e.g., temperature). Modeling environment attributes as discrete pose the following challenges:

- *State Space Explosion:* Some environment attributes can blow up the state-space. For example, an environment model for temperature with a granularity of 1F. Multiple apps together, using different environment attributes, exacerbate the problem.
- *Modeling Environment Effects:* Environment effects are challenging to model because actuator actions can affect each environment attribute differently, e.g. an A/C being turned on leads to a more significant temperature decrease than a fan.

To handle the first challenge, prior work models the critical states only [9, 23], which are the minimum and maximum values, or other values used in the app e.g., if an app turns A/C on when the temperature exceeds 76F, the temperature model will have three states for minimum and maximum values and for 76F. The critical states are not always obvious due to user-configurable inputs which leads to inexpressive models. Another solution is to increase the

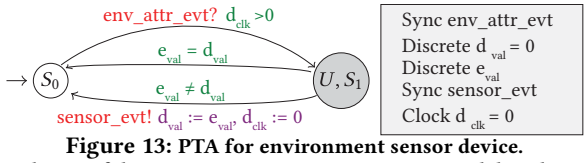


Figure 13: PTA for environment sensor device.

granularity of discretization, e.g., a temperature model with a step size of 5 will have states where temperature directly jumps from 60F to 65F. The larger the size of these steps, the less accurate the results. In PSA, we use this approach with a step size of 5.

To handle the modeling of environment effects, we can precisely model the effect of a device on the environment, e.g., if an A/C is turned on for 1 min, it causes a 2F decrease in temperature. But, this information is highly specific to the user's house, device vendor, etc. Another approach is to generalize the environment effects as either *increasing* or *decreasing* for non-categorical environment attributes, while for categorical attributes, there is an effect for each value e.g., *present* and *absent* for motion attribute. As a result, turning on the A/C and fan, both will cause a decreasing effect on the temperature model. PSA models environment effects using this approach, as we observe that this approach is enough to yield useful results.

Environment Metadata: PSA generates environment attribute models using this metadata. For a categorical attribute (e.g. motion), we define its states(e.g., present and absent) in the file. For non-categorical ones, we use domain knowledge to define their ranges (e.g., 60F-80F for temperature) (Appendix B.1).

PTA of Environment Attributes: Figure 12 shows the PTA for a non-categorical environment attribute. In the initial state *S_0*, the attribute value e_{val} can non-deterministically increase or decrease by a fixed *step*. On each value change, it synchronizes with devices that report on that environment attribute using sync (e.g., a thermostat reports on temperature). The environment attribute model can be influenced by the effects of *increase* or *decrease*. When *increase* effect is received, the model transitions to *S_1* where its value only increases. We add an invariant to each state so that the attribute value remains within $[min, max]$. This range is present in the environment metadata file. For example, for app model in Figure 9b, the action *AC_act* affects the temperature model (Appendix B.5) which will have *temperature_evt* as *env_attr_evt* in Fig 12.

Interaction Models: These map interactions of app actions on environment attributes and sensor models. For example, an interaction model will map *AC_act* sync with $a_{val} = 1$ for app in Figure 11 with the temperature model using *temperature_dec* sync (Appendix B.4). PSA generates these models using the *device-environment library* which maps devices actions to environment effects e.g., $A/C.on() \rightarrow temperature.decreasing$ (Appendix B.2). These effects are *increasing* or *decreasing* for non-categorical attributes, and *actual values* for categorical attributes. We manually built the library by looking at the specification of smart home frameworks and devices. It can easily be extended to interactions of new devices.

6.2 Modeling Devices

Devices in a smart home framework: 1) trigger apps that subscribe to events from these devices, 2) react to changes in the environment and app actions, 3) record the time when it changes its state, 4) support device state polling and triggered events. Our device models must support all these. To model the devices, we first categorize devices by surveying the smart home market and design separate

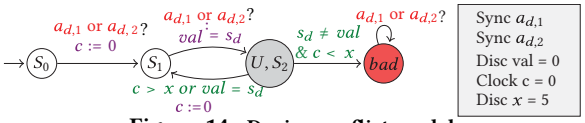


Figure 14: Device conflict model

PTA model templates for each type of sensor device. We categorize them as: 1) Environment sensor devices which are influenced by the environment e.g., temperature sensor. 2) Actuator sensor devices that are influenced by actuators either directly e.g., A/C as sensor, or indirectly e.g., a contact sensor for a door. 3) Independent sensor devices that are not influenced by environment attributes or actuators, e.g., presence sensor. Environment sensor devices and actuator sensor devices are modeled the same way.

Figure 13 shows the device model for environment sensor devices. In the initial state *S_0*, the model synchronizes on the sync *env_attr_evt* with the relevant environment attribute model e.g., a temperature sensor synchronizes with the temperature model using *temperature_evt*. At *S_1*, which is an *urgent* state (equivalent to having an invariant of $clock \leq 0$), if the updated value of e_{val} is not equal to current d_{val}, it updates d_{val}. It also triggers the app models using the sync *sensor_evt*. For example, the temperature sensor model will trigger apps that subscribe to it using *temperature_evt*. When the sensor updates its value, it also resets the clock d_{clk}, so that apps can use d_{clk} to get the time elapsed since the sensor changed state. To poll device state, an app model reads the value of the variable d_{val}. The guard $d_{clk} > 0$ on the S_0 to S_1 transition ensures that the sensor does not have multiple values in zero time. We show the thermostat model for app in Figure 4 in Appendix B.5. PTA models for independent sensors (e.g., presence sensor) non-deterministically keep switching states (e.g., presence active or inactive) which we do not show here for brevity.

Device Metadata: This file is used to generate device models. The file contains a list of device types as either environment sensor, actuator sensor, or independent sensor (Appendix B.3). For the environment/actuator sensor, we list the environment attributes/actuator devices that it reports on (e.g. temperature for temperature sensor, door for a contact sensor). For the independent sensor, we list its values (e.g., absent and present for presence sensor).

6.3 Modeling Intents

We use a well-known technique of using observers [5] to model safety properties. These are effect-free PTA models; i.e., they do not have any effect on the environment, sensor, and app models. The observer contains a special bad state that has no outgoing transition The verification then checks the reachability of this bad state. Now we model P1 and P3 to illustrate how we model complex safety properties as observers and show the rest in Appendix B.6, B.7.

Conflicting Actions: Consider two apps that issue actions $a_{d,1}$ and $a_{d,2}$ on device d. The observer model for device conflicts is shown in Figure 14. The model waits for $a_{d,1}$ or $a_{d,2}$ at S_0, and then measures the time between the next action. If the time is less than x and the two actions on d are conflicting, (checked by storing the value of the first action in *cur_val* and then comparing this value against the device state s_d when the second action happens), the model reaches a bad state. This model can be extended to include more actions from apps. Environment conflicts are also similar.

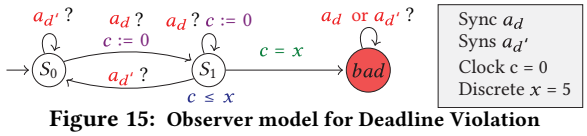


Figure 15: Observer model for Deadline Violation

Deadline Violation: Let $a_{d,}$ be an action on device d, and $a_{d'}$, be an action on device d', and let $a_{d,} \rightarrow a_{d'}$, be a time bound action, bounded by time duration x. Figure 15 shows the observer model for deadline violations, which transitions to a bad state if $a_{d'}$, does not happen within time x after receiving $a_{d,}$. This model can be extended to more actions and environment attributes.

7 Implementation

Our prototype implementation of PSA for the SmartThings platform has two main implementation components: the parser and the model generator. We implement our parser using Groovy. Our parser uses the abstract syntax tree (AST) of the app code for both of its passes. The parser visits AST nodes at the compiler's semantic analysis phase, where the Groovy compiler performs consistency and validity checks on the AST. Our implementation uses GroovyClassVisitor to extract the entry points and the structure of the analyzed app, and GroovyCodeVisitor to extract method calls and expressions inside AST nodes. The parser outputs PTA in a JSON format. Our model generator is written in Python. It takes the output of the parser and generates the required environment and device models. We use the IMITATOR PTA model checking tool [6] for its maturity. We express our models for the apps, intents, and environment interaction in the IMITATOR DSL.

8 Evaluation

We perform experiments on CloudLab nodes [16] with a 10-core 2.6GHz Intel E5-2660 processor and 160 GB of RAM. We use IMITATOR version 2.12. Our dataset has 86 SmartThings apps [33]. To build this dataset, we look at all the open source apps and exclude those which do not perform actuation, are integration apps between platforms (e.g. IFTTT and smartThings connect app), or if they use unsupported APIs (Section 5). These apps use sensors such as humidity, motion, water, temperature, illuminance, CO_2 and contact sensors, and door locks and electricity meter. These apps actuate on devices like door locks, cameras, vents, fans, lights, A/C, heater, HVAC, thermostat, garage doors, and valves. Table 3 summarizes the main features of these apps. These apps contain a mix of all app APIs discussed in Section 5 and hence are representative.

Total Apps	86
Apps with User Inputs	51 (59%)
Apps with Timed APIs	37 (43%)
Apps with Device Polling	33 (38%)
Stateful Apps	18 (21%)

Table 3: Summary of apps

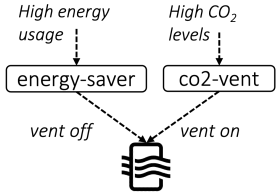


Figure 16: Example of a device conflict where a CO_2 vent is turned off when CO_2 levels are high

8.1 Findings

We run PSA on 86 apps to detect violations mentioned in Section 2.2. For conflicting actions, we run all apps in pairs, to see if they conflict with each other. We find 338 device conflicts (P 1.1), and 265 environment conflicts (P1.2). Around 95% of our device conflicts involve devices such as lights, A/C, and heater. Our indirect conflicts involve environment attributes such as temperature and humidity. We find 18 co-occurrence violations (P2), 8 deadline violations (P3), 4 blocked actions (P4) and 7 invalid action sequences (P5). Table 4a summarizes our results with some examples. Of the total violations, 23% can be avoided with safe input configurations e.g., the second example for P5 in Table 4a. Some of the violations we find are particularly dangerous. For example, Figure 16 shows a device conflict between the *co2-vent* and the *energy-saver* apps. The vent fan is turned on when the CO_2 level is too high. The energy saver app turns it off because the energy consumption crosses a threshold.

Ground Truth: To get ground truth for violations, we emulate the apps in SmartThings. For each violation that PSA finds, it outputs a trace in the form of a tree, where if the leaf node is a violation, we extract the path from root to that leaf node. This path gives us a series of events and time values that resulted in the violation. We use this sequence of dated events, and play them in SmartThings to reproduce the violation. SmartThings allows users to define virtual devices that are controlled by programmable device handlers. To feed event sequences output by PSA to real apps in this virtualized environment, we write scripts to generate custom device handlers. These device handlers generate fake events as specified by the PSA output. We compare the log trace of SmartThings with PSA to obtain ground truth. We also test for false negatives (Section 8.3). However, testing for all possible scenarios is very hard in case of no violation, specially because of configurable user inputs.

8.2 Comparison with baselines

In Section 2, we argue that not modeling the stateful and timed behavior of apps can lead to false positives and false negatives. Now, we quantitatively compare PSA against two baselines. The first baseline does not model state and models timed events but not duration in apps (B1). The second baseline models discrete state but not time in apps (B2). Both of these do not model indirect environment-device interactions such as heater *on* action will increase temperature, causing thermostat reading to rise. B1 represents SiFT [23], and B2 represents Soteria [9]. Since prior efforts do not directly apply to our specific setting, these baselines are approximations of prior work. Table 4b summarizes our results.

As discussed before, PSA outputs a set of safe configurations. We separately verify PSA output (Section 8.3) for correctness. For testing against the baselines, we use pre-configured apps. To configure an app for a property, we pick an input value from the set of safe configurations output by PSA and one value from unsafe configurations. For cases where no safe configurations exist, we pick a random value to configure that app. This approach allows us to identify false positives and false negatives. We get the ground truth by using the same approach we described in Section 8.1. Except here for non-violations, we play the apps with the chosen configuration to see if the violation happens. We define false positives (for a given app configuration) when the violation does not exist but the verification tool outputs a violation. A false negative is when the verification tool returns no violations, when in fact one does exist. Table 4b highlights the false positive (FPR) and false negative (FNR) rate for B1, B2 and PSA for all the 86 apps. The last column highlights if we can reproduce the violation using the counter-example

Prop.	Total	Examples	Apps involved in Example
P1.1	338	(ii) CO_2 vent is turned off when CO_2 levels are too high (iii) Door is locked and unlocked at the same time	co2-vent, energy-saver unlock-it-when-I-arrive, lock-it
P1.2	265	(i) Vent fans to dehumidify, and humidifier are on (ii) A/C and heater are turned on at the same time	auto-humidity-vent, humidifier turn-it-on-when-im-here, Its-too-cold
P2	18	(i) Lights and A/C are turned on when there is no one in the house (ii) Heater, thermostat, etc., are turned off when the user is at home	my-light-toggle, turn-off-with-motion switch-changes-mode, keep-me-cozy
P3	8	(i) Door remains unlocked for more than 5 min (ii) Lights are not turned off within 10 min of no motion	unlock-when-I-arrive light-up-the-night
P4	4	(i) Water sprinklers cannot turn on because the water supply is off (iii) Thermostat set-points cannot be set because thermostat is off	close-the-valve, sprayer-controller keep-me-cozy, thermostat-auto-off
P5	7	(i) Too many same notifications sent to the user very close in time (ii) Lights are turned on and off frequently causing a strobing effect	text-me-when the-flasher

(a)

Prop.	BL	FPR	FNR	CE
P1.1	B1	0.35	0	0.19
	B2	0.27	0	0.39
P1.2	B1	0.21	0	0.22
	B2	0.16	0	0.41
P2	B1	0.11	0	0.28
	B2	0.6	0	0.45
P3-5	B1	0	1	0
	B2	0	1	0
	PSA	**0**	**0**	**1**

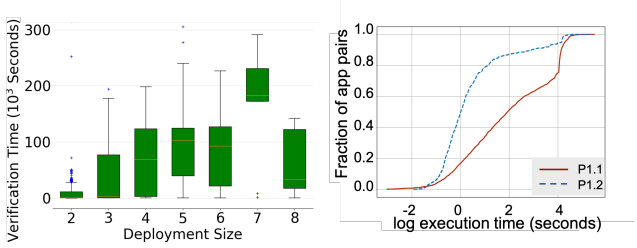
(b)

Table 4: Summary of our findings in 4a. Comparison with baselines in 4b (Prop. = property, BL = baseline, CE = counter-example)

provided by the tool. For P1 and P2, PSA reduces false positives by up to 35%. In cases, where we drop state and timed behavior of apps, the counter-examples also lacks that information, which makes it hard to reproduce that violation in B1 and B2. Properties P3 and P5 cannot be verified without modeling time. Hence, a 100% false-negative rate depicts that tools representative of B1 and B2 cannot find any such violation. For property P4, both B1, and B2 cannot find any violation that PSA finds because they do not model indirect environment interactions (e.g. by turning the thermostat off, setting heating setpoint will not have any effect).

8.3 Validating PSA

We use a multi-pronged validation approach as discussed below.
Test Apps: To check the correctness of PSA for false negatives, we create a representative set of test apps that cover all APIs of SmartThings (Table 2). For each test app A, we generate A' such that it has the same logical structure, but its output(device action) is opposite to that of A such that they result in a device conflict (Appendix C.1). We find that PSA successfully points out all conflicts.
Real Apps: To check for false positives, we consider all violations output by PSA for device conflicts. For each violation, we randomly sample 10 paths in the analysis trace that lead to a violation. We replicate them in our SmartThings virtualized environment. If we observe the violation, we declare it as a true positive, else mark it as a false positive. We did not find any false positives. To test for false negatives, we input a random sample of 100 app pairs to PSA, and check for violations. In case of no violation, we generate 50 random event sequences of length 10 with an arbitrary duration (0s to 1000s) between each event to trigger apps. We run these random event sequences in our virtualized smart home environment to check if we find a violation (Appendix C.2). We did not find any false negatives. We understand that this testing methodology is by no means exhaustive. Testing for all possible scenarios is very hard, specially because of configurable user inputs. Hence, we randomly pick samples and test on those to get some confidence on our results.
Correctness of Models: To evaluate the correctness of models, we feed an event sequence to real apps in the virtualized environment, and compare the logs to the output trace of PTA models (Appendix C.3). For each app, we generate 50 event sequences of length 10. We introduce arbitrary (0s to 1000s) duration between events. If the output trace of the models is the same as SmartThings logs, then our app model is consistent with the app. We find that app models generated by PSA are consistent with real apps.



(a) (b)

Figure 17: Time to verify a given deployment size for P1 (a). Variation in verification time across apps for P1.1 and P1.2 (b).

8.4 Performance of PSA

We denote the number of apps in a smart home as the "deployment size". Then, from our 86 smartThings app dataset, we sample 100 combinations at random for a given deployment size (n=2 to n=8) and report the time to verify these apps for a given violation as shown in Figure 17a. The time for the analysis to complete increases exponentially with an increase in deployment size. The significant variations in the box plot highlight the varying complexity of real apps. For a given deployment size, if the random set generated involves less complex apps, the time for analysis is also small. Figure 17b shows how the computation time varies for a fixed deployment size. If an app has too many event handlers, timed APIs, user inputs, then the analysis will take a lot of time.

9 Related Work

We discussed the key prior work [9, 14, 23, 28, 31] in Section 2. Here, we discuss other related efforts.
IoT Security: Many existing works formally analyzing IoT devices [2, 27] but do not model application behaviors and environment interactions. Croft et al., looks at control programs such as home automation and SDNs and models them as timed automata [13]. It does not model the interaction between devices, environment and apps, and does not handle user-configuration. Similarly, Alhanahnah et al., models app interactions in a smart home but skips state and time in apps [1]. Many prior works also focus on run-time conflict detection and resolution [15, 17, 24, 25], and static conflict detection [29, 36] which assume stateless-TAP rules. Our work considers a broader class of violations than these for complex stateful-TAP apps. DepSys [28] also finds conflicts in apps at static time, but it requires the developers to define the triggers and actions of the app in a metadata file, and does not model app behavior. IoTGuard [10] and Wang et al., [38] instrument apps

to check for safety properties at run time, which is complementary to our work. IoTMon [14] marks smart home apps' interaction as risky or safe, by looking at their triggers and actions. It does not model the internal app behavior and cannot reason about violations that PSA finds. Similarly, Helion [26] uses statistical language modeling to generate home scenarios to automatically define safety intents. This work is complementary to PSA.

CPS and Verification: Using timed automata (TA) to model cyber-physical systems is not new. Kumar et al. [21] models industrial automation systems using TA. Sun et al., uses TA to formally verify an aerial video tracking system [35]. Croft et al. use it to formally verify control systems [13]. We build on this work and make it practically useful for modeling smart home apps.

10 Discussion and Conclusion

The interaction of apps in smart homes can cause safety violations. We are not the first to identify this issue; however, we note that prior efforts have key expressiveness issues in tackling the timed, stateful, and parameterized behavior of apps and their complex interactions. To overcome these limitations, we design PSA which uses PTA to model a smart home. We evaluate PSA on 86 SmartThings apps. We compare PSA against two baselines. We find 19 previously missed intent violations and have 35% fewer false positives. We discuss the limitations and future directions for PSA below:

Intents Library: Our Intents library can be extended by the community or the smart home vendor. Prior works like Helion [26] can also be used to auto-generate intents based on home data.

Risky vs. Benign violations: Some violations discovered by PSA maybe benign. To screen violations, there is complementary work [14] which try to categorize app interactions as risky or not.

Scalability: As we envision PSA to be used offline by the smart home vendor, the high verification time should not be a problem. In the future, we intend to prune our models by removing the app interactions irrelevant to the property using taint analysis. We are also looking at parallel verification approaches [6] for PTA.

Need for Source Code: Similar to prior work [9, 31], PSA requires app code which is reasonable if vendor deployed. However, we can use blackbox modeling techniques to infer app behavior ([14]). Such models will be noisy but they do not require source code.

Usability: We plan to conduct user studies to improve the output of PSA so that non-expert users can understand and take action.

Environment Models: Discrete modeling of environment attributes requires no manual effort in PSA. However, PSA requires a domain expert to define more complex environment models.

Acknowledgments

We thank our shepherd Kishore Ramachandran, and all reviewers for their comments. We acknowledge the contributions of Yifei Yuan and Anand K. Prakash in helping with the project. This research has been supported in part by NSF awards TWC-1564009, SaTC-1801472 and the CMU Cylab IoT initiative.

References

[1] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. 2020. Scalable analysis of interaction threats in iot systems. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 272–285.

[2] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. Sok: Security evaluation of home-based iot deployments. In *S&P'19*. IEEE, 1362–1380.

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

[4] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. 1993. Parametric Real-time Reasoning. In *STOC '93*. ACM, New York, NY, USA, 592–601.

[5] Étienne André. 2013. Observer patterns for real-time systems. In *ICECCS'13*. IEEE, IEEE, 125–134.

[6] Étienne André. 2021. IMITATOR 3: Synthesis of timing parameters beyond decidability. In *CAV*. Springer, 552–565.

[7] RBoy Apps. 2019. http://smartthings.rboyapps.com/

[8] Lei Bu, Wen Xiong, Chieh-Jan Mike Liang, Shi Han, Dongmei Zhang, Shan Lin, and Xuandong Li. 2018. Systematically ensuring the confidence of real-time home automation IoT systems. *ACM TCPS'18* 2, 3 (2018), 1–23.

[9] Z Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated iot safety and security analysis. In *USENIX ATC'18*. USENIX, 147–158.

[10] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *NDSS*. USENIX.

[11] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2020. Cross-app interference threats in smart homes: Categorization, detection and handling. In *DSN'20*. IEEE, IEEE, 411–423.

[12] Edmund M Clarke and E Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*. Springer, Springer, 52–71.

[13] Jason Croft, Ratul Mahajan, Matthew Caesar, and Madan Musuvathi. 2015. Systematically exploring the behavior of control programs. In *ATC 15*. USENIX.

[14] Wenbo Ding and Hongxin Hu. [n.d.]. On the safety of IoT device physical interaction control. In *CCS'18*. ACM, ACM, 832–846.

[15] Colin Dixon et al. 2012. An operating system for the home. In *NSDI'12*. USENIX.

[16] Dmitry Duplyakin et al. 2019. The Design and Operation of CloudLab. In *USENIX ATC*. USENIX, 1–14.

[17] Emre Göynügür et al. 2017. Policy conflict resolution in iot via planning. In *Canadian Conference on Artificial Intelligence*. Springer, Springer, 169–175.

[18] Thomas A Henzinger. 2000. The theory of hybrid automata. In *Verification of digital and hybrid systems*. Springer, 265–292.

[19] Hubitat. 2020. Private Home Automation. https://hubitat.com/.

[20] IFTTT. 2020. IFTTT. https://ifttt.com/.

[21] Pratyush Kumar, Dip Goswami, Samarjit Chakraborty, Anuradha Annaswamy, Kai Lampka, and Lothar Thiele. 2012. A hybrid approach to cyber-physical systems verification. In *DAC'12*. IEEE, IEEE, 688–696.

[22] Chieh-Jan Mike Liang et al. 2016. Systematically debugging IoT control system correctness for building automation. In *ACM BuildSys'16*. ACM, 133–142.

[23] Mike Chieh-Jan Liang, Börje Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Lucis Li, and Yong Yu. 2015. SIFT: Building an Internet of Safe Things. In *IPSN'15*. ACM.

[24] Meiyi Ma et al. 2016. Detection of runtime conflicts among services in smart cities. In *2016 IEEE SMARTCOMP*. IEEE, IEEE, 1–10.

[25] Meiyi Ma et al. 2017. Cityguard: A watchdog for safety-aware conflict detection in smart cities. In *IoTDI'17*. 259–270.

[26] Sunil Manandhar et al. 2020. Towards a natural perspective of smart homes for practical security and safety analyses. In *IEEE S&P'20*. IEEE, 482–499.

[27] Mujahid Mohsin et al. 2016. IoTSAT: A formal framework for security analysis of the internet of things (IoT). In *2016 IEEE CNS*. IEEE, IEEE, 180–188.

[28] Sirajum Munir et al. 2014. DepSys: Dependency Aware Integration of Cyber-Physical Systems for Smart Homes. In *ICCPS '14*. IEEE, 127–138.

[29] Alessandro A Nacci et al. 2018. BuildingRules: A Trigger-Action–Based System to Manage Complex Commercial Buildings. *ACM TCPS'18* 2, 2 (2018), 1–22.

[30] Julie L Newcomb et al. 2017. IOTA: a calculus for internet of things automation. In *ACM SIGPLAN'17*. ACM, 119–133.

[31] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. 2018. IotSan: fortifying the safety of IoT systems. In *CoNEXT'18*. ACM, ACM, 191–203.

[32] José Reinaldo Silva and Pedro MG del Foyo. 2012. Timed petri nets. In *Petri Nets: Manufacturing and Computer Science*. InTech, 359–378.

[33] SmartThings. 2019. Repository. https://github.com/SmartThingsCommunity/.

[34] SmartThings. 2020. Documentation. http://docs.smartthings.com/en/latest/.

[35] Youcheng Sun, Giuseppe Lipari, and Étienne André. 2015. Verification of two real-time systems using parametric timed automata. (2015).

[36] Yan Sun, Xukai Wang, Hong Luo, and Xiangyang Li. 2014. Conflict detection scheme based on formal rule model for smart building systems. *IEEE Transactions on Human-Machine Systems* 45, 2 (2014), 215–227.

[37] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. 2019. Charting the attack surface of trigger-action iot platforms. In *CCS'19*. 1439–1453.

[38] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and Logging in the Internet of Things. In *NDSS 2018*. USENIX.

[39] Zapier. 2020. Connect your apps and automate workflows. https://zapier.com/.

[40] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In *2019 ICSE*. IEEE, 281–291.

A Modeling Choices

The Table 5 summarizes our modeling choices across three dimensions: 1) Time, 2) User Inputs, 3) Environment Attributes.

Environment	Time / User Input	Discrete	Continuous
Discrete	Enumeration	FSM	Timed Automata [4], Timed PetriNets [32]
	Parameterization	Symbolic Automata	Parameterized Timed Automata [4], Parameterized Timed PetriNets [32]
Continuous	Enumeration	-	Hybrid Automata (HA) [18]
	Parameterization	-	Parameterized HA [4]

Table 5: Choices for modeling abstractions for smart home

B Smart Home Modeling

We give details regarding our smart home modeling. We first provide example of environment metadata in Appendix B.1, device environment library in Appendix B.2, and device metadata in Appendix B.3. Then, we give example of an interaction model in Appendix B.4. We also give a high-level picture of how all the models interact in Appendix B.5. Finally, we show the PTA models for properties P4 and P2 in Appendix B.6 and B.7.

B.1 Environment Metadata

Examples of a few entries in the environment metadata is shown in Table 6. For categorical environment attributes e.g., motion, we also list their states.

Table 6: Environment Metadata

Environment Attributes	States
Carbon Monoxide	clear, detected, tested
Carbon Monoxide	number
Motion	present, absent
Smoke	clear, detected, tested
Sound	detected, clear
Temperature	number
Air Quality	number
Illuminance	dark, light
Illuminance	number
Energy Usage	number
Dust	present, absent

B.2 Device-Environment Library

We give examples of a few entries in the device environment library in Table 7. For each device type and an action, the library lists environment effects for the attributes that it affects.

B.3 Device Metadata

We give examples of a few entries in the device metadata. For each sensor device, we list its type. In case of an independent sensor device, we list its states. In case of device and environment sensor devices, we report the device/environment attribute it reports on.

B.4 Interaction Model

Figure 18 shows the interaction model that maps the actions of A/C on the temperature environment model using temperature effects of *temperature_inc* and *temperature_dec*.

Table 7: Device-Environment Library Example

Device Type	Action	Effect
Door	open	contact.open
Door	close	contact.close
Alarm	ring	sound.detected
Light	on	illuminance.increasing
Light	off	illuminance.decreasing
AC	on	temperature.decreasing
AC	off	temperature.increasing
TV	on	power.increasing
TV	on	sound.detected

Device	Type	Details
Presence Sensor	Independent	detected, not detected
Contact Sensor	Device	Door
Motion Sensor	Environment	motion
Shock Sensor	Independent	present, absent
Smoke Detector	Environment	smoke
Sound Sensor	Environment	sound
Heater	Device	heater
Water Sensor	Environment	water
Valve	Device	Valve
Dust Sensor	Environment	Dust
Illuminance Measurement	Environment	illuminance
AC	Device	AC

Table 8: Device Metadata Examples

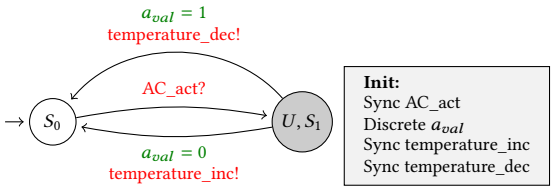


Figure 18: Interaction model that maps actions of A/C on temperature model.

B.5 Interaction between all models

We illustrate how all the models interact with each other in Figure 19. The temperature model synchronizes with the thermostat model using *temperature_evt*. If the temperature value e_{val} has changed, the thermostat model synchronizes with the app model using *thermostat_evt*. The app issues an *AC_act* which is synchronized with the interaction model, and the device conflict model. The interaction model then issues temperature effects *temperature_inc* or *temperature_dec* depending on the AC state a_{val}.

B.6 Blocked Action Violation

Consider $a_{d,}$ be an action on device d, and $a_{d'}$ be an action on device d'. The observer model to check if $a_{d,}$ blocks $a_{d'}$ is shown in Figure 20. The model transitions to S_1 when $a_{d,}$ is received, and then it transitions to the state *good* if $a_{d'}$ is received. Then we check the reachability of the *good* state, indicating that $a_{d'}$ is not blocked by $a_{d,}$.Note that in addition to checking if $a_{d'}$ can happen after $a_{d,}$, we also need to check if $a_{d'}$ can happen at all. This will require another model which transitions to *good* state if $a_{d'}$ happens.

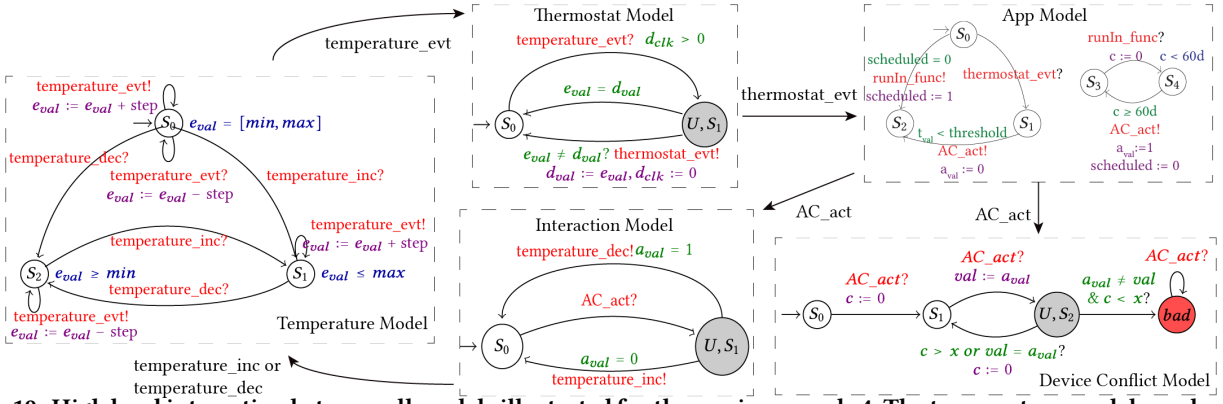


Figure 19: High level interaction between all models illustrated for the app in example 4. The temperature model synchronizes with the thermostat model using *temperature_evt*. If the temperature value e_{val} has changed, the thermostat model synchronizes with the app model using *thermostat_evt*. The app issues an *AC_act* which is synchronized with the interaction model, and the device conflict model. The interaction model then issues temperature effects *temperature_inc* or *temperature_dec* depending on the AC state a_{val}.

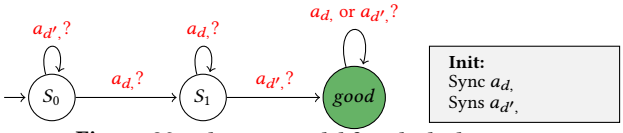


Figure 20: Observer model for Blocked Actions

B.7 Co-occurrence Violation Model

To implement Co-occurrence violation 2 in PSA, the observer model listens for device events, to see if the required actions happen together. For instance, to check $smoke\_sensor.detected \rightarrow alarm.on()$, the observer model listens for smoke sensor events, and if the value is detected, then it checks if $alarm.on()$ action is issued. Let evt^d denotes an event from device d, and s_d denotes the device state, then observer model to detect that action $a_{d'}$, happens when the device state s_d of device d is equal to y is given in Figure 21

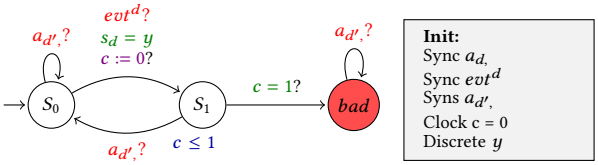


Figure 21: Observer model for Co-occurrence violation

C Testing

Now we illustrate how we validate PSA for correctness.

C.1 Testing using Test Apps

Figure 22 shows the methodology for checking false negatives in PSA using test app pairs which are logically equivalent but perform the opposite device action leading to a conflict.

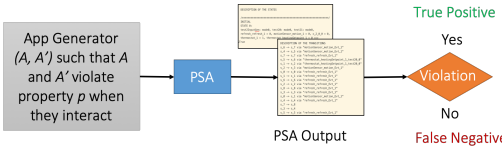


Figure 22: Testing methodology for checking if PSA has any false negatives using app pairs for which we know the ground truth.

C.2 Testing using Real Apps

Methodology for checking the correctness of PSA output is shown in Figure 23. In case of a violation, we play the counter-example in the SmartThings virtualized environment and compare the logs with PSA output. In case of no violation, we generate random event sequences and emulate in SmartThings to reproduce the violation.

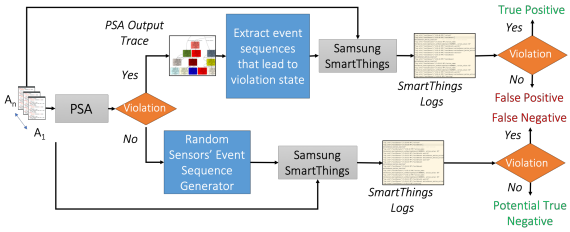


Figure 23: Checking the correctness of PSA output.

C.3 Correctness of Models

We illustrate our methodology to check for the correctness of our generated PTA models for the apps in Figure 24. For each app, we feed random event sequences to the PTA model of the app, and play the same event sequences in SmartThings to trigger the app. Then we compare the SmartThings logs with the PSA analysis trace.

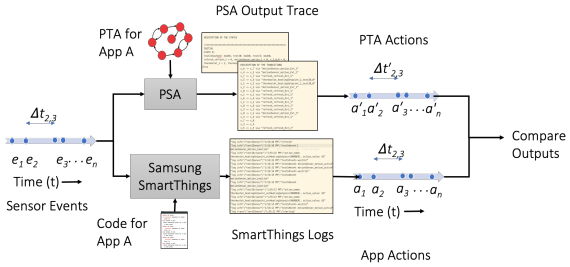


Figure 24: Checking the correctness of PTA models.