# DIAC: An Inter-App Conflicts Detector for Open IoT Systems

XINYI LI, Chang'an Univerisity, China
LEI ZHANG, North Carolina State University, USA
XIPENG SHEN, North Carolina State University, USA

This paper tackles the problem of detecting and solving potential conflicts among independently developed apps that are to be installed into an open Internet of Things (IoT) environment. It provides a new set of definitions and categorizations of the conflicts to more precisely characterize the nature of the problem, and proposes a representation named IA Graphs for formally representing IoT controls and inter-app interplays. Based on the definitions and representation, it then describes an efficient conflicts detection algorithm. Combining conflict categories, seriousness indicator and conflict frequency, an innovative solution policy for solving various detected conflicts is developed, which also takes into account user preference and interest by providing interactive process. It implements a compiler and runtime software system that integrates all the proposed techniques together into a comprehensive solution. Experiments on SmartThings apps validate its significantly better detection efficacy over prior methods and effectiveness of conflict solution with user preference.

CCS Concepts: • **Computer systems organization → Embedded systems**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: IoT, Compiler, Conflicts Detection

**ACM Reference Format:**
Xinyi Li, Lei Zhang, and Xipeng Shen. 2019. DIAC: An Inter-App Conflicts Detector for Open IoT Systems. 1, 1 (March 2019), 24 pages. https://doi.org/10.1145/1122445.1122456

## 1 INTRODUCTION

IoT (Internet of Things) is a network of physical objects, devices, vehicles, electronic components, software, and sensors that connect to each other and exchange data [15]. It works with the help of internet, sensors, and IoT devices. Many existing IoT platforms are special-purpose platforms. The IoT system in the engineering field is used for the management of production, marketing, safety and delivery. The IoT system in a hospital field makes it easy for doctors to monitor patients and diagnose illness remotely. These IoT platforms are called *closed IoT platforms*, in which the whole system is designed coherently and all the apps and operations are predefined by a group of expert developers to conform to a certain model.

Unlike the closed IoT platforms, open IoT platforms allow the public to develop and upload IoT apps which can be downloaded and deployed by other users in their IoT environments. Therefore in an *open IoT platform*, the apps are not predefined and may be developed by independent developers without any cooperation or interaction. Open IoT platforms are becoming increasingly influential, exemplified by the fast development of home automation systems (e.g.,

Authors' addresses: Xinyi Li, Chang'an Univerisity, Xi'an, China, xinyili@outlook.com; Lei Zhang, North Carolina State University, Raleigh, North Carolina, USA, lzhang45@ncsu.edu; Xipeng Shen, North Carolina State University, Raleigh, North Carolina, USA, xshen5@ncsu.edu.

SmartThings from Samsung [26], HomeOS [9]). On SmartThings, any developer may write an app in SmartThings SDK, and upload it to SmartThings cloud. Any user can then download the app from the cloud, install and deploy it in his or her SmartThings environment. In an open IoT environment, there are many deployed apps interacting with the sensors and devices in the same environment. Because these apps are independently developed, unexpected interplays among them often happen, resulting in undesirable issues, of which one is conflict. In an open IoT environment deployed with an energy-saving app `ESave` and a security app `SafeHouse`, `ESave` turns off a light when no motion is detected in the last 2 minutes, while `SafeHouse` tries to simulate the presence of people in a house by making the light stay on and off alternatively periodically (half an hour for each state) when the house is empty. Because there is a conflict between the two apps, `SafeHouse` fails to simulate the presence of people in the house as the light in the house remains off most of the time.

Our examination of the apps on Samsung SmartThings app repository [25] shows that such inter-app conflicts are common. Among 113 commodity SmartApps, we found 73 groups of conflicting apps (Section 7). The number of smart connected homes could rise to 700 million by 2020 [10], and about 1.5 million IoT developers are currently working on Smart Home projects [28]. With more developed apps and used sensors, the inter-app conflict problem will become even more serious.

Conflicts could exist in closed reactive systems (e.g., an automobile control system). But because the platforms are closed and the set of modules and their interactions are defined by a team in a coordinated manner, the problem has been addressed through a specification-based method [3]. The development of these systems often starts with some formal specifications of the design (e.g., through Esterel-like synchronous languages [2]), which then gets translated into actual implementations in either software or hardware. A primary concern in those systems is how to validate whether the implementation is consistent with the specification rather than detecting the conflicts among foreign apps [22]. The method does not apply to open IoT platforms as users may install arbitrary apps into them.

There have been several efforts trying to address inter-app conflicts on open IoT platforms, but they only considered some basic types of conflicts: two apps access the same device (e.g., in HomeOS [9]) or have different/opposite effects on the same device (e.g., in SIFT [21]) or environment (e.g., DepSys [23]).

In this work, we argue that the previous definitions of inter-app conflicts are over simplified, and subsequently, the conflict detection methods are inadequate. We provide an improved definition that better aligns with users experience and intuition (validated through a user study), categorize the conflicts into strong conflicts and weak conflicts, and further divide the weak conflicts into three subcategories depending on their different natures.

The fourth part of this paper presents our challenging conflict detection method. It proposes a graph structure named *IA Graph* to concisely represent the controls in each IoT app and the various schedules of events. Based on the representation, it uses an efficient algorithm to leverage first-order logic and SMT solvers to detect conflicts of all types.

Innovative conflict solution for various conflicts is then designed. Seriousness indicator is introduced to reflect the potential risk levels of device capabilities involved in conflicts, and conflict frequency expressing the occurrence time of conflicts is defined. Combining conflict categories, seriousness indicator and conflict frequency, an innovative solution policy for solving various detected conflicts is developed, which also takes into account users' preference and interest by providing interactive process.

We develop a tool named DIAC (for Detector of IoT App Conflicts), which integrates all our proposed techniques together, along with a compiler for automatically constructing IA Graphs from SmartThings apps. DIAC supports different IoT platforms by design and is applicable in various IoT environments.

The experiments demonstrate that DIAC can successfully detect and solve the conflicts among those apps with marginal time overhead. Compared to results from previous works, DIAC shows significantly improved precision (88%) and recall (100%).

This paper extends a LCTES' 19 conference paper [19] by the same authors in the following major aspects.

*Refining weak conflicts.* Weak conflicts are not always unacceptable to users and their seriousness is often subjective. Categorizing them into three classes makes the heuristic conflict solution solves the conflicts automatically and taking users' acceptance and preference into account at the same time.

*Redefining conflict frequency and introducing seriousness indicator.* Conflict frequency is redefined as the probability product of the two controls involved in a conflict, more reasonably reflecting the occurrence frequency of a conflict. Seriousness indicator is introduced to express the seriousness level and user acceptance of a capability. The two metrics can help the automatic and interactive execution of the developed heuristic conflict solution, combining the influence degree of conflicts and user preference.

*Developing a heuristic conflict solution.* A heuristic conflict solution policy is developed, which deals with all the detected conflicts combing conflict natures and user selection based on the showed detailed conflict information, favorably avoiding unintended changes especially when a smarthome has many apps deployed and devices used.

*Updating DIAC.* Adding a component implementing the developed heuristic conflict solution to the infrastructure of the original DIAC in [19], enabling the updated DIAC to systematically analyze multiple SmartThings apps and their interplays, automatically detect possible conflicts, heuristically solve all the detected conflicts by taking into account their natures and users preference based on the showed detailed information in specific scenarios.

*Evaluating the updated DIAC with more apps.* Using more apps with more device types and capabilities to evaluate the performance of the updated DIAC in detecting and solving possible conflicts. Unlike the original DIAC only with detecting capability, the updated DIAC can not only detect conflicts existing in more device types and capabilities, but also deal with the conflicts in a heuristic way by combing conflict natures and user intentions to provide enhanced safety level and better user acceptance.

## 2 BACKGROUND AND RELATED WORK

This section describes background that is necessary for understanding the rest of the paper and some related works.

*IoT Programming.* Ways to write an IoT app are in general in two categories: rule-based, scripting language-based. The former is represented by IFTTT [14], in which, app developers just write a few high-level rules. The latter is represented by Samsung SmartThings [26], in which, app developers write apps in an SDK on scripting language `Groovy` [12]. Rule-based methods are easy to use by general users, while scripting languages are Turing complete, offering flexibility in programming a broader range of apps.

The main techniques developed in this work are about the principled issues on inter-app conflicts. They are hence applicable to IoT apps in both categories. We implement and test them on SmartThings: Its extra flexibility over rule-based methods helps expose the full challenges.

*SmartThings.* SmartThings provides a cloud-based platform as Figure 1 shows. It offers an SDK for common programmers to develop apps, called *SmartApps*, that can run in a SmartThings environment to control compatible devices.

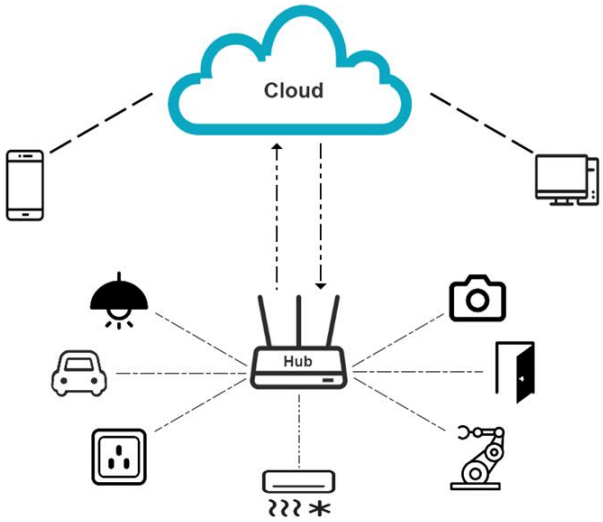


Fig. 1. SmartThings Architecture

SmartApps are installed by the user via the SmartThings mobile client application. SmartThings offers a rich toolset to develop, test, and publish custom code. Everyone can publish SmartApps on SmartThings cloud; it is free for users to download SmartApps from the cloud. Once a SmartApp is installed on a mobile device (e.g., a smartphone connected with SmartThings cloud and environment), the selected IoT devices can be accessed and controlled by the SmartApp. The communications between all connected devices and the cloud and mobile apps are supported by SmartThings Hub which connects directly with the broadband router. SmartApps may execute in the SmartThings cloud, or on the hub.

SmartThings SDK is based on scripting language Groovy [12], equipped with libraries special to SmartThings. Figure 2 shows the code of a simple SmartApp.

The "definition" segment is for meta data of the app. The "preferences" segment defines what kinds of devices are required by the app and other options. When a user installs this app on her smartphone, the app will pop up a dialogue window, in which, it lists all the devices with the "contactSensor" capability in the user's SmartThings environment, and ask the user to pick the one the user would like the app to control. After that, the dialogue window will ask the user to pick the device with the "switch" capability in a similar manner. This installation process automatically binds the to-be-controlled physical devices with the variables ("contact1" and "light1" in our example) in the app.

The third part includes some predefined methods that are automatically called during SmartApp installation, updating, and deletion. For our example app, the method "initialize" is called when the app is installed. That method uses the "subscribe" mechanism in Groovy to define the event handler (method "openHandler") as the method to call when the status of device "contact1" changes to "contact.open". In SmartThings, when a device with a "contactSensor" capability opens, its status in the SmartThings runtime turns to "contact.open" automatically. The "updated" method is called when the preferences of an installed app is updated. The "unschedule" method removes all scheduled executions for this app. SmartThings also provides the "schedule" API to create recurring schedules. In our example, "offHandler" is called every day at the time "scheTime" specified by the user.

```
//Metadata that determines how the app is described
//in the mobile app UI along with other options.
definition(
name: "A Sample SmartApp",
namespace: "demo",
author: "Demo User"
description: "Turn a light on when a door opens."...
)

//Defines what devices and other options are required.
//Drive the installation screens in the mobile app UI.
preferences{
section("Select devices"){
input "contact1", "capability.contactSensor",
                    title: "Select contact sensor"
input "light1", "capability.switch", title: "Select a light"
input "scheTime", "time", title: "Time to execute every day"
}}

//Pre-defined methods that are called during
//SmartApp installation and updating.
def installed(){initialize()}
def updated(){
    unsubscribe()
    unschedule()
    initialize()}
def initialize(){
    subscribe(contact1, "contact.open", openHandler)
    schedule(scheTime, offHandler)}

//Event handlers specified in event subscriptions and
//other methods required to implement the SmartApp.
def openHandler(evt){
     light1.on()
     runIn(60*5, offHandler)}
def offHandler(){light1.off()}
```

Fig. 2. A SmartApp.

The final part includes the definition of other methods of the app. In our example, this part contains the definition of the method "openHandler" and "offHandler". Method "openHandler" first sends a request to change the status of device "light1" to "on". Upon an invocation of the "onHandler" method, the cloud sends such a request to the SmartThings Hub, which then sends a signal to the physical device that has been bound with the variable "light1" and that device will then turn on its switch. The second call in "onHandler" is "runIn" method, which is a SmartThings API that invokes a method after a certain time. "60*5" denotes the number of seconds from the invocation time of "runIn" to call method "offHandler". The call of "runIn" in the example invokes method "offHandler" 5min after the call of the "runIn" method. "offHandler" sends a request to change the status of device "light1" to "off". The mechanism behind this request is the same as "light1.on" in "onHandler".

In our discussion, we use *event* to represent that a device is actuated. An event could be triggered by an entity external to an application (e.g., sensor input, mobile phones, human operations), or internally by another event specified in the app code. All events are asynchronous.

This simple example illustrates a set of operations happening behind the scene enabled by the SmartThings platform: the creation of the dialogue for users to input their device selections and other options, the binding between physical devices and variables in the code, the invocations of the pre-defined methods, and the materialization of the effects of a device status changing request, which include all the needed communications and the interactions with the physical devices. A challenge for developing a tool to analyze inter-app conflicts is how to implement, emulate, or circumvent these functionalities. We will come back to this point in Section 6 while presenting the implementation of our DIAC platform.

*Inter-app conflict in IoT systems.* Recent years have seen increasing interest in home automation and other IoT applications. The work closely related with this study includes HomeOS [9], SIFT [21], and DepSys [23]. HomeOS [9] is one of the first Operating Systems designed for home automation. It gives only brief discussions on inter-app conflicts, regarding two apps conflicting if they access the same device at the same time. The detection happens through runtime monitoring of device assesses by apps. This definition is imprecise and is subject to some important limitations. SIFT [21] proposes a rule-based programming platform for IoT. It gives some valuable discussions on the safety of IoT apps including inter-app conflicts. The definition used in SIFT [21] is as follows: two apps conflict if they try to cause different (and incompatible) actions on the same device simultaneously. However, due to the preliminary definitions of inter-app conflicts that SIFT uses, its conflicts detection method has a limited coverage. DepSys [23] investigates the conflicts when multiple cyber-physical systems are integrated into a smart home. It introduces the concepts of priority and emphasis to help resolve conflicts. Two apps conflict if (a) they may access the same device at the same time or (b) they access different devices whose direct effects on a certain aspect of the environment are different. This definition, to a large degree, resembles what DepSys [23] uses. Its definition still misses some important classes of conflicts. Moreover, DepSys treats the app as a blackbox and employs no source code analysis. It requires the developers to provide specifications on the set of devices each app accesses and their effects, which impairs its practical usability. An IA-Graph based conflict detection algorithm is provided in [19]. It tackles the problem of detecting potential conflicts in open IoT system. The detection of conflicts is validated better efficacy than previous work [9, 21, 23]. However, the focus of [19] is on conflict detection, how to resolve conflicts is not discussed.

*Detection of event-driven systems.* There are some recent studies on helping with the debugging of IoT control system correctness [7, 20]. They concentrate on the implementation correctness of a single IoT rather than inter-app conflicts. They require the users to provide some specifications on the desired control policies and then use model checking methods to detect the inconsistencies between the implementation of an app and the policies. The problem focused in this work is inter-app control conflicts. Debugging within an app [7, 20] is outside the scope of this work.

IoT is a kind of event-driven system. There have been a number of studies on detecting concurrency bugs in event-driven programs by monitoring memory accesses [13]. They concentrate on data races, rather than inter-app conflicts.

There have been some recent work on detection of functionalities and security issues of IoT systems. SmartAuth performs static analysis of the source code and uses natural language processing techniques to analyze code annotations, capability requests of apps and developers' descriptions in the app store, to detect whether an app's functionalities faithfully follow the expectation of users [27]. ContexIoT proposes a context-based permission system for IoT platforms to identify fine-grained context information and prompt the information at runtime for users to make an access control decision [16]. Tyche introduces a risk-based permission system to solve the over-privilege problems by grouping permissions according to their risk levels and allows users to grant permissions for a device by specifying an allowed

risk level [11]. HOMEGUARD applies symbolic execution to extract rules from apps and employs a constraint solver to evaluate the relation between rules for systematic threat detection [6]. The problem discussed in this work only focuses on the safety of IoT system.

*Conflicts in real-time/reactive systems.* Conflicts could happen in traditional real-time/reactive systems [8, 18]. They are typically closed in the sense that the set of modules and their interactions in the systems are defined in the design stage of the systems by a team in a coordinated manner. The development of these systems often start with some formal specifications of the design (e.g., through Esterel-like synchronous languages [2]), which then gets translated into actual implementations in either software or hardware. A primary concern in those systems is how to validate whether the implementation is consistent with the specification. Our targeted IoT platform is open, allowing the public to install downloaded arbitrary apps in their IoT environments. The primary concern on such platforms is hence different: Rather than checking for consistency against specifications, we are trying to detect inter-app conflicts. As there are no specifications for each user's IoT environment, the solution naturally differs as well.

*Resource conflicts in open systems.* Some common computing systems are also open (e.g., apps are allowed to be installed into a Linux workstation or Android phone). There are resource conflicts in these systems (e.g., multiple apps request disk or memory). They are different from the inter-app conflicts in our work. The conflicts in those systems are mostly about enabling fair and efficient sharing of common resources or finding data races or deadlocks among threads [1, 4, 17]. The problems tackled in this work are about detecting whether one app's control on a device capability gets affected by another app.

## 3 DEFINITIONS AND CATEGORIZATIONS OF INTER-APP CONFLICTS

A proper definition of inter-app conflicts is fundamental for inter-app conflicts detections. This section examines the limitations of the definitions used in prior work, and then presents our definitions and categorizations of conflicts.

The three definitions used in previous studies (including HomeOS [9], SIFT [21], DepSys [23]) on detecting inter-app conflicts form an evolving path, getting increasingly more sophisticated. For instance, definition in HomeOS [9] is quite rudimentary and would even inappropriately regard readings of the same sensor by two apps as a conflict. Definition in SIFT [21] avoids that issue, while definition in DepSys [23] goes one step further; it considers some conflicts by two different devices. An example is that one app turns on a dehydrator while the other turn on a hydrator in the same room. Even though they turn on different devices, they still create conflicting effects.

Despite the progress, the refined definition leaves out an important class of conflicts: the cases when one app affects the conditions used by another app as triggers for some actuators. In *SecHouse Example*, a security app `SecHouse` starts video taking when the room is dark and a door contact is open, while a home assistant app `MiniAid` turns on the light whenever the door contact is open. In this example, the two apps control different actuators; one is video camera, the other is light. These two devices do not have direct effects on the same aspect of the environment, and hence do not fit the previous definitions of conflicts. However, when these two apps are deployed together, `SecHouse` will not function normally as `MiniAid` disables the video taking action of `SecHouse` by preventing the needed conditions from being met.

The definition of inter-app conflicts in this work addresses the issues of the previous definitions. It meanwhile classifies conflicts into several categories, which could help discriminative treatment to the conflicts of different seriousness.

Specifically, we define inter-app conflicts in two main categories: strong conflicts and weak conflicts. Below we try to give intuitive but informal definitions of them first, and will provide more rigorous definitions in the next section.

Definition 3.1. **Strong Conflict:** *When multiple apps run together, there is a strong conflict when some actions of an app get disabled as a result of an interaction with the other apps.*

Here, an "action" is an act a device takes (e.g., a light turns on, a camera takes a shot). The two apps in the *SecHouse Example* in the previous subsection form a strong conflict. When the two apps in the example run together, `SecHouse` will not function normally as `MiniAid`, by turning on the light, disables the video taking action of `SecHouse` by preventing the needed conditions from being met.

Definition 3.2. **Weak Conflict:** *When multiple apps run together, there is a weak conflict if the apps control an actuator differently while no actions of an app get disabled.*

An example of weak conflict is the `SafeHouse` and `ESave` example we mentioned in the introduction. Because the control of the lights is changed by `ESave` (turning off the lights if no motion is detected in 2 minutes), `SafeHouse` fails to simulate the presence of people in the house.

The final phrase in the definition excludes strong conflicts from weak conflicts. In weak conflicts, no actions of the apps are disabled. In our example, `SafeHouse` still turns on and off the lights as it is programmed. However, the control of the lights gets affected by the other app and hence changes the resulting effects of `SafeHouse`.

*Implicit Conflicts.* In both examples we have given, the conflicts happen to the control or actions on the same functionality of the same device that multiple apps operate on. There are situations in which two apps operate on different devices but still form conflicts. An example is two apps that control two different lights in the same room. They both affect the brightness or the color of light in that room. Detecting such implicit conflicts needs some ways to capture the implicit relations between devices and between device capabilities. We use *influence zone* to address the issue as Section 4.1 will present.

Together, these definitions cover all conflicts that we have encountered in our examinations and experiments, including all the aforementioned conflicts that previous definitions cannot cover. In general, strong conflicts are problematic as they reduce the functionalities of some apps, while it is less clear for weak conflicts as we will see in the next section. It is worth noting that the seriousness of a *conflict* relates with the involved devices, and is sometimes also subject to user's expectations. The definition of *weak conflicts* captures all control-related conflicts (except strong conflicts). Some *weak conflicts* could be fully tolerable for a user in some scenarios. Section 4.3 will elaborate this point and explain how the detection algorithm refines *weak conflicts*. The categorization helps us design detection algorithms suitable for each type of the conflict. It can also potentially help with discriminative solutions of different types of conflicts. Section 5 will concentrate on effective conflict solution by taking user's expectations into account.

## 4 CONFLICTS DETECTION

When the definition of conflicts is narrow, the detection can be simple. For instance, the detector needs to check only the set of devices each app accesses as HomeOS does [9]. As the definition of conflicts become more comprehensive, the simple detection method becomes inadequate. More complicated interplays between apps need to be examined and analyzed, which calls for more sophisticated designs of the conflict detection method.

Our designed conflict detection method consists of two key components. The first is the representation of the key events and relations of the apps. The second is the automatic inferences upon the representation to detect and report conflicts. As conflicts depend on the relations between different capabilities of the same or different devices, the detection requires modeling of these relations. To assist the reasoning, we design a graph representation to concisely

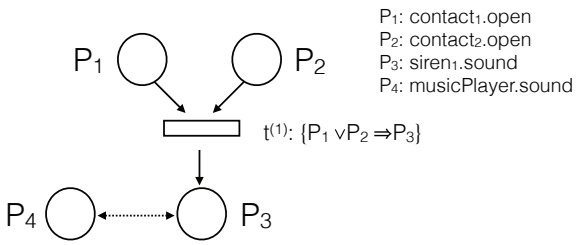


Fig. 3. A simple IA Graph.

capture all possible interplays among apps and their controls of the devices. Based on the representation, we construct an automatic inference algorithm to detect and report all conflicts. This section first presents the representation and then describes the inference algorithm.

### 4.1 Representation: Inter-App Graphs (IA Graphs)

The first step in the conflict detection is to use a concise, easy-to-reason formalism to represent the key events and conditions of all the SmartApps installed in an IoT system. Such a formalism should meet three conditions.

- It should be flexible enough to express the operations of various devices, the conditions for these operations to get triggered, and the consequences of the operations.
- It should capture all the essential interplays of different apps in a single IoT system, including those implicit ones (which, as aforementioned, arise even when two apps control different devices.)
- It at the same time should be amenable for automatic inferences of all possible conflicts between apps.

Our solution is *IA Graphs*. We first provide a formal description, and then illustrate it with an example.
An IA Graph is denoted as $IAG = (V_p, V_g, V_s, T, E_c, E_i)$. Its six elements are defined as follows.

- $V_p$: a set of *capability vertices*. A capability vertex is a node that represents a capability of a device. Its ID is set to be the concatenation of the device ID and the capability. For instance, a capability vertex, "contact.open", represents the open capability of a contact device.
- $V_g$: a set of *global variable vertices*. A global variable vertex is a node that represents a global variable in a SmartThings app. Its ID is set to the concatenation of the app name and the variable name. Such variables are used to remember information across function invocations. Representing them explicitly in IA Graphs makes it possible to consider their influence on the behaviors of the apps.
- $V_s$: a set of *schedule status vertices*. A schedule status vertex is a node that represents the scheduling status of a function in a SmartThings app. Some functions in a SmartThings app are made to run at certain times through the "schedule" API. We call them *scheduled functions*. Each *schedule status vertex* corresponds to a scheduled function in an app. The vertex has two attributes: "flag" and "schetime". Attribute "flag" has two values, *true* if the function is scheduled, and *false* if unscheduled. Attribute "flag" changes its value once "schedule" or "unschedule" is called. Attribute "schetime" records the schedule time of the function (which is the first variable in "schedule" API.) Such vertices are essential for the treatment of time-related conflicts as shown later.
- $T$: a set of *transition function vertices*. Each transition is a collection of Horn clauses, with each in the implication form of "conditions $\Longrightarrow$ consequences", where both "conditions" and "consequences" are some logic clauses,

with the former indicating the conditions under which the transition takes place and the latter indicating the consequences of the transition. We call those "conditions" the *Preconditions* and those "consequences" the *Postconditions* of a transition node. Vertices involved in *Preconditions* form the *Preset* of the transition, while vertices involved in *Postconditions* form the *Postset* of the transition. If a capability $p$ belongs to both Preset and Postset of a transition, $p$ and $p'$ denote that capability in Preset and Postset respectively. Each transition carries an app ID such that when multiple apps in an IoT environment are put together into one IA Graph, the transitions from different apps can be easily told apart.

- $E_c$: a set of *control edges*. Each control edge is a directed edge that flows between a vertex in $V = V_p \bigcup V_g \bigcup V_s$ and a vertex in $T$. The sources of *control edges* flowing into a transition vertex (i.e., a node in $T$) form its *Preset*, and the targets of *control edges* coming out of a transition vertex form its *Postset*. Formally, a control edge flowing from vertex $s$ to vertex $t$ is represented as $< x, y >$, where, $x$ and $y$ must meet the following conditions: $x, y \in V \bigcup T$, and if $x \in V$, then $y \in T$, and $x$ must be part of the Preset of $t$; if $x \in T$, then $y \in V$, and $y$ must be part of the Postset of $x$. These edges represent the control relations among the vertices.
- $E_i$: a set of *influence edges*. Each influence edge is a bi-directional edge connecting two capability vertices. It is related with *influence zone*, a concept we introduce. *Influence zone* refers to the attribute of an area that is physically influenced by a capability of a device. For instance, the switch of a light in a room influences the brightness in that room; the brightness of that room is the influence zone of that switch. An influence edge connects two capability vertices if and only if the two capabilities have overlapped influence zones. Such edges help in detecting implicit conflicts. Section 4.3 will further elaborate this concept when describing device models.

*Example.* Figure 3 provides a simple example IA Graph. It contains four *capability vertices*, depicted in circles, $P_1, P_2, P_3, P_4$, representing the "contact" capability of two contact sensors, the "sound" capability of a siren and a music player. The box $t^{(1)}$ in the middle of the graph represents a transition vertex. It indicates that when either contact1.open or contact2.open is true, the siren should sound. The superscript of $t^{(1)}$ indicates the ID of the app that contains this transition function. The dotted line between $P_4$ and $P_3$ in Figure 3 illustrates the influence edge between the two vertices as their influence zones are both the sound of the same room. The other edges are control edges, showing the control relations among the device capabilities.

### 4.2 Construction of IA Graphs

This part describes the derivation of IA Graphs from SmartThings apps. It is potentially extensible to other IoT programming languages.

The construction process is overall based on code analysis and the semantics of SmartThings APIs. Particular complexities exist in the derivation of the Horn clauses for transition functions, especially on how to model time-related relations and how to deal with control flows in the apps. In this section, we first provide a brief description of the overall construction process, and then focus on those special complexities.

*4.2.1 Overall Process.* The constructor is written based on the compiler inside the open-source Groovy engine [12], parsing the source code of input SmartThings apps and creating vertices and transitions of IA Graph accordingly. Static code analysis is used in identifying the control statements relevant to device controls and creating symbolic notions for them. Effects of scheduled functions are determined through the static analysis of the code in the functions and the device modeling. The transition functions are derived through the static analysis of the relevant device control

Table 1. Models of Some SmartThings APIs for IA Graph Construction

| API with Description | Preset | Postset | PreConditions | PostConditions |
|---|---|---|---|---|
| *subscribe(erDev, "capability.value", subFunc)*: Subscribe to event, i.e. when the status of *erDev* is changed to *capability.value*, actions in *sunFunc* will take place as *sunFunc* runs | the vertex denoting the capability of the triggering device *erDev* | vertices denoting the capabilities of the triggered devices in *subFunc* | *"erDev.value"* | change the status of the triggered devices as per *subFunc* |
| *schedule(scheTime, scheFunc)*: Execute *scheFunc* every day at *scheTime* | vertices denoting the triggers of "schedule" | the vertex denoting *scheFunc* | the status of the triggering devices | *"scheFunc.schetime = scheTime & scheFunc.flag = true"* |
| | the vertex denoting *scheFunc* | vertices denoting the capabilities of the triggered devices in *scheFunc* | *"scheFunc.flag = true"* | Set the status of triggered devices and set the "chronos" of triggered devices as *scheFunc.schetime* |
| *unschedule(scheFunc)*: unschedule the function *scheFunc* | vertices denoting the triggers of "unschedule" | vertex denoting *scheFunc* | the status of trigger devices | *"scheFunc.flag = false"* |
| *runIn(runInTime, runInFunc)*: execute *runInFunc* after *runInTime* seconds from now | vertices denoting the triggers of "runIn" | vertices denoting the capabilities of the triggered devices in *runInFunc* | the status of trigger devices | Set the status of the triggered devices and set the "chronos" of triggered devices as the sum of "chronos" of trigger devices and *runInTime* |

statements and the modeling of SmartThings APIs. The constructor models the effects of both branches of a conditional statement.

Specifically, when an app is installed in the system, vertices are created for devices, scheduled functions and global variables; transitions are constructed based on data flow and control flow analysis. The generation of IA Graphs focuses on the controls of devices and device triggering relations in the apps. As most device controls are set through SmartApp APIs (e.g., "subscribe", "schedule", "unschedule", "runIn"), we build some models to capture the semantic of the main SmartThings APIs, which simplifies the code analysis. Table 1 gives examples of the transition construction rules for some APIs. There are other scheduling methods that create recurring schedules, such as "runOnce", "runEvery5Minutes", "runEvery1Hour". The only difference among these methods is the recurring frequency. The transition construction rules for these APIs are similar to those of "schedule" and "runIn".

*4.2.2 Addressing Complexities in Time, Branches and Loops.* How to handle time-related operations is essential for IoT systems. In our context, it has two-fold issues.

The first is on how to model the effects of time-related APIs. An example is "schedule(scheTime, scheFunc)". An invocation of the API sets a function ("scheFunc") to execute every day at a particular time ("scheTime"). Our solution is to explicitly model the effects of such APIs in the constructor. The third row in Table 1 shows the Preset, Postset, PreConditions, and PostConditions, of that API. The bottom two rows in Table 1 show the models of another two time-related APIs.

The second fold of complexity is on how to represent temporal relations in Horn Clauses. Temporal relations have two categories: those on *absolute time*, and those on *relative time*. The former are temporal constraints of one vertex, such as "when time is between 8pm and 11pm, light turns on if contact is open"; the latter are temporal relations between events, such as "when the door gets closed, turn off the light after 5 minutes". Our solution for representing such relations is inspired by the treatment to time in classic high-level Petri Nets [5, 24]. We equip each vertex in an IA Graph with a special property named "chronos" to represent the triggering time of the vertex. And meanwhile, each IA Graph is considered to have an invisible (virtual) vertex "chronos" representing the current time. This virtual vertex has invisible edges reaching all other vertices. Together they enable representations of time-related events in the transition functions. For instance, the aforementioned *absolute time* example can be expressed as "contact.open

& 8pm≤ contact.open.chronos≤ 11pm $\implies$ light.on"; the *relative time* example can be represented as "chronos == door.closed.chronos + 5min => light.off=1". This feature plus the aforementioned representations of scheduled functions equip IA Graphs with the flexibility for representing time-related aspects of an IoT system, giving the foundation for the detection of time-related conflicts.

For a condition statement, the capabilities of devices appearing in its branches are represented as nodes in IA Graphs as those in other statements. The only special aspect is that the expressions checked by the condition statement become part of the presets of the transitions of those vertices. Loops are rarely used in SmartApps. Repetitive actions are usually materialized by "schedule" kind of APIs. Symbolic notions are used to express the impact of loop on device control. For example, if trigger $x$ lets a camera take photos repeatedly for 4 times with a 5min delay in the between, the transition is written as $\{x \implies camera.picTaking = 1; chronos = camera.picTaking.chronos + i \times 5 \times 60 (i = [1, 2, 3]) \implies camera'.picTaking = 1\}$.

Some computations produce numerical values used to set the parameters of some devices (e.g., temperature for a thermometer). In the generated IA Graphs, when the compiler cannot figure out the exact values, it uses symbolic notions (i.e., free variables) to represent these computation results in the transition functions.

IA Graphs offer a way to represent the controls of multiple apps in a single form. The connections in IA Graphs naturally manifest the control dependences within each app and the interplays between apps. It offers the conveniences for a conflict detector to identify the parts of the controls in each app that are relevant to the controls of certain common device capabilities, and ignore the irrelevant parts, as shown next.

### 4.3 Inference for Detecting Conflicts

In this section, we describe the algorithm for detecting inter-app conflicts. We first introduce a term *solution set*.

*Solution Set.* The detection works on the IA Graph of apps. Recall that each transition in an IA Graph carries a function which is the conjunction of a set of implication formulas. These formulas can be converted into a conjunction norm through first-order logic operations. For instance,

$switch = 1 \implies light = 1$ turns into $light \vee -switch$,

where, "light" and "switch" are boolean variables.

For a given transition $t$, let $S$ be the complete set of assignments to the variables in $t$ that could satisfy the function of $t$. We call $S$ the *solution set* to that transition, denoted as $S(t)$. For the previous example, $S$ contains two sets of solutions: {light=1, switch=∗} and {switch=0, light=∗}, where ∗ represents all possible values. For a subgraph of an IA Graph, the solution set to the conjunction of all the transitions in that subgraph is *the solution set of that subgraph*.

*Operational Definitions.* Before presenting our detection algorithm, it is necessary to first give some more rigorous definitions of the conflicts. The definitions are operational, preparing for our detection algorithm designs.

Definition 4.1. Operational Definition of Strong Conflicts: *For a set of apps A, let $F$ be the conjunction of all the functions contained in A, and $S$ be the solution set to $F$. A contains a strong conflict if and only if there is at least one solution (denoted as $s$) to some app in A that can not be implied by $S$, that is, $S \not\Longrightarrow s$.*

The consistency of this definition and the informal one in Section 3 is intuitive: This definition essentially says that after putting the apps together, the set of behaviors allowed for one of the apps becomes smaller than before—that is, some actions are disabled. For instance, consider two apps: `t1: switch=1` $\implies$ `light=1` in app1, `t2: switch=1` $\implies$ `light=0` in app2. Function $F$ would be $(light \vee -switch) \wedge (-light \vee -switch)$. First-order logic simplifies it into $-switch$.

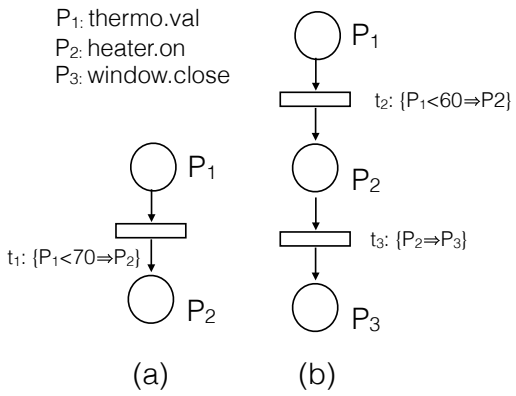


Fig. 4. Two apps with weak conflicts on $P_2$.

The solution set $S$ is hence $\{switch = 0, light = *\}$. It does not imply one of the solutions to app1: $\{switch = 1, light = 1\}$. The two apps have a strong conflict.

Before describing the operational definition of weak conflicts, we introduce a term *control subgraph* of a vertex. For a vertex $p$ in an IA Graph, its control subgraph is the subgraph of the IA Graph, in which, every vertex can reach $p$ along at least one path. We denote the subgraph with $cg(p)$. In Figure 4 (b), for instance, $cg(P_3)$ is the entire graph, while $cg(P_2)$ contains only $P_1$, $P_2$, and $t_2$. The control graph of a vertex captures how the capability in that vertex is controlled.

Definition 4.2. Operational Definition of Weak Conflicts: *For a set of apps $T$, there is a weak conflict if there is a vertex $p$ such that $S_i(p) \neq S_j(p)$, where, $S_i(p)$ and $S_j(p)$ are the solutions of the control graphs of $p$ in apps $a_i$ and $a_j$; $a_i \in T$ and $a_j \in T$, and they both involve $p$ in their control.*

We illustrate the intuition of the definition through the two example apps shown in Figure 4. Vertex $P_2$ appears in both IA Graphs. Its control graphs in the two apps are respectively the entire IA Graph in Figure 4 (a), and the "$P_1$, $t_2$, $P_2$" part of the IA Graph in Figure 4 (b). The solution sets of the two control graphs are respectively {heater.on, -(thermo.val<70)} and {heater.on, -(thermo.val<60)}. These two differ, which indicates the differences in the control of $P_2$—which is the condition for a weak conflict in our informal definition given in Section 3. The IA Graph representation provides conveniences for identifying the relevant control subgraphs and omitting the irrelevant controls for the conflict inferences.

*Refinement of Weak Conflicts.* Weak conflicts have a broad range, including all cases where the controls of a device differ in two apps. They are not always unacceptable to a user. For instance, two apps both control a siren, one sounds it when smoke is detected and the other sounds it when a window is open. For most users, this weak conflict is no issue.

It would help if weak conflicts of different seriousness could be separated by the detection process. However, a perfect automatic solution is in principle extremely difficult if ever possible. The seriousness of a conflict is often subjective. For the aforementioned siren example, even though most users do not mind of the conflict, some users do if they want the siren to be a dedicated smoke detector and dislike the ambiguity the second app introduces.

Our algorithm hence still captures all weak conflicts, but at the same time, tries to indicate heuristically whether a weak conflict is likely to be an issue for most users. The key insight underpinning this refinement is the connections with user acceptability. Specifically, we categorize conflicts into the following three classes:

Table 2. Models of Influence Zones of Some Devices

| Capability | Influence zone | | Type of conflicts |
|---|---|---|---|
| | area | attribute | |
| light.switch | room | brightness | weak |
| thermo.switch | room | temperature | weak |
| musicplayer.play | room | sound | weak |
| robot.cleaningmode | room | space | weak |
| auto-wheelchair.movingmode | room | space | weak |

- (C1) Conflicts for exclusive control. For these conflicts, two different controls usually create unfavorable interferences. Examples include heater.on, cooler.on, hydrator.on, dehydrator.on.
- (C2) Conflicts for shared control. For these conflicts are usually intended for multiple uses; different controls on them are often acceptable. Besides siren.sound, other examples include SMS.sendMessage, phone.call.
- (C3) Others. The likelihood for a user to accept such conflicts is less clear than C1 and C2. An example is light.on. In our `SafeHouse` and `ESave` examples, the different controls of light.on cause `SafeHouse` to malfunction. But for another two apps (`sunsetApp` and `earlyRiseApp`) which require the light to turn on respectively at sunset and in early morning, the different controls could be both wanted by a user. Other examples include lock.lock, stereo.play.

*Device Modeling for Influence Zone.* As we have mentioned, conflicts may occur even between different devices if their influence zones overlap. To help detect such conflicts, we build a set of models to characterize the influence zone of a capability of a device. An influence zone consists of the area that the capability affects and the attribute of the area that was affected. Table 2 lists the models of some capabilities of some common devices. For each influence zone, the table also indicates the type of conflicts that may be caused due to an overlap of the zone by two apps; some are weak and some are strong. For instance, turning on a light influences the brightness of a room, and two lights in the same room form some weak conflicts as both may cause changes to the same attribute of the room. Take another example, two moving devices influences the available space of a room. A robot cleaner setting in the cleaning mode occupies the space of a room, and an auto-wheelchair setting in the moving mode also takes up the space of the same room, such two moving devices in the same room form weak conflict as both may occupy the same space of the room.

### 4.4 Detection Algorithm

We design an algorithm that uses the two operational definitions and device models to detect all types of conflicts. It is for installation-time detection, working when an app is to be installed into a system that already holds some other apps.

The algorithm first adds the new app (denoted as $b$) into the IA Graph. It then finds all the vertices of $b$ that appear in some existing apps, which we call *common vertices*. Each element in `InfSets` is a set, consisting of the vertices that have overlapped influence zones with one vertex of $b$. Then, for each *common vertex* in $b$, it gets the logic formula in the control graph of the vertex in every app. If any of them is not equivalent to $b$'s, a weak conflict is reported on that vertex as per Definition 4.2. It then checks whether the conjunction of all the formulas of a *common vertex* is compatible with each of the formulas; if not, it regards that a strong conflict as per Definition 4.1. It then checks each set in `InfSets`. If it contains vertices from multiple apps, it checks the type of the influence zone. If it is weak, the set forms some weak conflicts; if it is strong, it checks whether there are two different vertices from that set belonging to different apps and the corresponding capabilities can take place at the same time. If so, it regards it as a strong conflict.

The detection of strong conflict checks whether there is a solution of $f_2$ that cannot be implied from any solution of $f_1$—that is, whether some function of $f_2$ is disabled by $f_1$. The check is done through an SMT solver (Z3 [29]) by checking whether $f_2 \wedge (-f_1)$ can be satisfied. If so, it means that there is a solution that makes $f_2$ true but $f_1$ false. There hence is a strong conflict. (To avoid interferences from irrelevant variables, before the check, variables appearing in neither the postset nor the preset of $f_2$ are removed from $f_1$.)

The algorithm involves the proving of about $2\sum_{i=1,|P|} n(i)$ formulas, where $|P|$ is the number of *common vertices* and $n(i)$ is the number of apps sharing the $i^{th}$ *common vertex*. Because typical IoT apps are small, each formula involves only several IoT device capabilities and hence can be proved easily by enumerating all possible combinations of the firing capabilities. And usually a vertex is shared by just several apps in a collection. As a result, the complexity does not appear to incur much overhead as our experiments shows.

Moreover, in practical usage, apps are often installed incrementally. Incremental conflict detection is hence a more common usage. It only needs to check whether the newly added app conflicts with existing ones. It takes even less time to do.

*Discussion on Cooperations.* One might wonder whether intended cross-app cooperations would be marked as conflicts by the algorithm. In SmartThings, we have seen cooperations among different modules in one app, but not among different apps. To our knowledge, common app developments of SmartThings put all the intended device controls into one app rather than ask users to install multiple separate apps to get the cooperative control. With that said, it is not difficult for our solution framework to handle such cooperations: The cooperative apps could be labeled as a group and be regarded as a single entity in our analysis.

## 5 CONFLICTS SOLUTION

This part mainly discusses the conflict solution. Conflict detection is designed to detect all possible conflicts. Strong conflicts are often more serious than weak conflicts as they disable some functionalities of an app. But conflict solution ultimately depends on the user and the specific context and scenario. Since different users have different intentions and preference even if they install the same apps with the same set of devices, further decisions and operations from users are hence required to resolve conflicts.

As a result, conflict solution typically involves an interactive process. Our conflict detection method may offer several-fold help for the interactive process. It can provide a list of inter-app conflicts, and indicate each as a strong or weak conflict. Thanks to the design of its detection algorithm, it can further provide more detailed information on the conflicts. To help with discriminative solutions of different types of conflicts, we introduce two metrics: seriousness indicator and conflict frequency.

### 5.1 Solution Metrics

*Seriousness Indicator.* In addition to conflict frequency metric, to express the seriousness nature of a conflict, another term called seriousness indicator is provided for the device capability. We perform a user study with 28 participants from our institutions to develop seriousness indicator distinguishing different capabilities. Participants are given all of the remaining SmartThings supported device capabilities with detailed explanation of related commands and attributes, except for those capabilities with exclusive or shared control. Participants are then asked to measure the seriousness of different capabilities with three choices: low, medium and high. Low level of seriousness capability has no commands or has commands without potential risk, medium level of seriousness capability has commands affecting warning

Table 3. Examples of Seriousness Indicators for Different Capabilities

| Capability | Command | Seriousness Level |
|---|---|---|
| Alarm | both,off,siren,strobe | Medium |
| Carbon Dioxide Measurement | None | Medium |
| Door Control | close,open | High |
| Garage Door Control | close,open | High |
| Lock | lock,unlock | High |
| Refrigeration Setpoint | setRefrigerationSetpoint | Low |
| Robot Cleaner Cleaning Mode | setRobotCleanerCleaningMode | Low |
| Signal Strength | None | Low |
| Smoke Detector | None | Medium |
| Window Shade | close,open, presetPosition | Low |

apparatus (e.g., alarm and oven) but without life or property threatening risk, and high level of seriousness capability has commands posing great threats to the safety of life and property (e.g., door and lock). The seriousness level of capability is regarded as the seriousness indicator of the device capability. Examples of seriousness indicator are showed in Table 3.

*Conflict Frequency.* First, we introduce the frequency of a control. It is the maximum number of times that the control may take place in a day. The conflict frequency of two controls of device capabilities is the product of the frequencies of the two controls.

The frequency is calculated through simple linear algebra on the temporal conditions included in the preconditions of transitions. A more complete solution is to estimate the actual frequency by considering all preconditions; it is much more complicated. In cases where our tool cannot figure out the frequency of a conflict, those conflicts are shown as a " frequency unknown" group. SmartThings supports 71 device capabilities which define a set of commands and attributes that devices can support. Based on that, users bind smart devices to SmartApps at the app installation time.

### 5.2 Heuristic Conflict Solution Policy

To solve the conflicts, we propose a heuristic policy. The solution is determined by the conflict category, seriousness indicator and conflict frequency.

For a strong conflict, the installation of a new app is rejected and the information on which functionality of which app could get disabled is provided.

For a weak conflict, we take the two metrics including seriousness indicator and conflict frequency into consideration to offer different approaches to solve it.

- For C1 conflict, two different controls create unfavorable interference, and this kind of conflict is unacceptable by users. The heuristic policy gives the information on which apps have different controls on which device capabilities. And the conflict frequency is also displayed to users. These information could potentially help users make better decisions when selecting the options to resolve the conflicts, including uninstalling an app, disabling an app, or ignoring this conflict.
- For C2 conflict, shared controls are often acceptable, installation of a new app is performed and the detailed information about which apps have different controls on which device capabilities is displayed to users. The solution policy for C2 conflict differs by the seriousness indicators of the device capabilities involved in the conflict. If the seriousness indicators are in the same level, the two apps are executed and the conflict is ignored. Otherwise if the seriousness indicators belong to different levels, the app with the higher lever indicator is executed and the app with the lower level indicator is disabled.

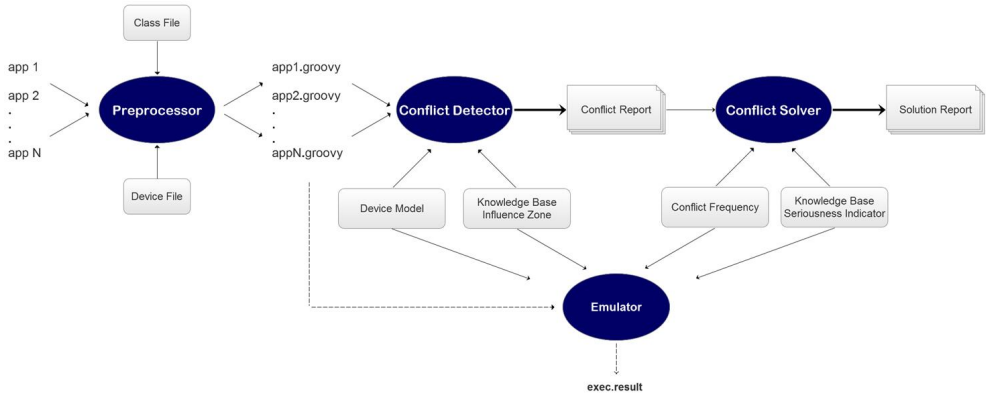


Fig. 5. Infrastructure of DIAC

- For C3 conflict, the installation of a new app is performed and the detailed information about which apps have different controls on which device capabilities is displayed to users. The solution policy for C3 conflict varies according to the seriousness indicators of the device capabilities involved in the conflict. If the seriousness indicators fall into different levels, the app with the higher level indicator is executed, guaranteeing users' interest. Since the likelihood for a user to accept the conflict is unclear with the same seriousness indicator level, further decisions from users are hence required to resolve the conflict. With the conflict frequency provided, users can select options to resolve the conflict by disabling any one of the apps or ignoring the conflict.

To our knowledge, this is the first proposed solution that distinguishes conflicts of different seriousness. Its effectiveness is validated through a user study in Section 7. We do not aim to provide a perfect automatic detection of conflict seriousness—which may not be possible without user explicitly expressing their expectations and preference, but to offer a simple approach to assist users in handling inter-app conflicts. It is worth noting that even for C2 and some "non-serious" C3 conflicts, detecting and showing them can still be useful. By showing users the changes to the old controls of devices, that can help users avoid unintended changes—especially when a smart home has many apps deployed and the users may not clearly remember all the controls.

## 6 DIAC SYSTEM

We develop a software system named DIAC (for Detector of IoT App Conflicts) that incorporates all the proposed techniques. It consists of some extra features to serve as an system to allow automatic analysis of multiple SmartThings apps and their interplays. As the SmartThings platform is not open-source, it is not directly usable. DIAC is based on an open-source Groovy Engine [12].

Figure 5 shows the high-level infrastructure of DIAC. It consists of the following four major components, represented by the four ovals.

(1) *Preprocessor.* This component takes an original SmartApp as its input and transforms it into a form amenable for the open-source Groovy Engine to execute without changing its original meaning.

As mentioned in Section 2, executions of a SmartApp relies on the behind-scene support (e.g., binding physical devices with variables in an app, materializing device status changes) of a set of libraries in the SDK. The SDK is not open-source. To circumvent the problem, we create a set of classes to emulate the functionalities of those supports. These classes include *Capability classes*, *DeviceType classes*, *CapabilityGUI classes*, *CapabilityCollection classes*, and *System classes*.

*Capability classes* are used to describe capabilities of devices. Instances of capability classes are initialized in *DeviceType classes* since one device has one or several capabilities. *CapabilityGUI classes* are used to create virtual devices. There are buttons in GUIs, offering the virtual interface for users to interact with the apps while testing them. *CapabilityCollection classes* are used to bind variables and capabilities in apps. *System classes* are designed to provide some system services.

Overall, these classes empower DIAC to realize the binding between physical devices and variables in the code, the invocations of the pre-defined methods and the materialization of the effects of a device status changing request. The *Preprocessor* links the input SmartApp with these customized classes, making the apps compilable and executable by the standard Groovy Engine.

As user installing a SmartApp, the *Preprocessor* maps the physical devices to virtual objects in the platform. Some environmental information like *area* is also added to the analyzer. For example, DIAC is able to map the light bulb to *bedroom area* when a new light is installed in a user's bedroom. This information will be helpful for further analysis of conflicts caused by *influence zone*. For example, if two lights are connected in the same room, the *area* information is necessary to analyze the control of brightness of the room.

(2) *Conflict Detector.* This component takes multiple preprocessed SmartApps as input, derives the IA Graphs through the modified Groovy compiler, and then applies the conflict detection algorithm to detect all the conflicts. It works upon a device model library that we have developed, which captures the key relations among the different states of some common capabilities of devices. To classify weak conflicts, the system is also equipped with the ability to distinguish different effects caused by controls on devices. For example, 'heater.on' is often treated as a more exclusive control than 'SMS.sendMessage'.

Both the model library and the detection algorithm are described in the previous sections. Besides, we construct a knowledge base for the models of Influence zones, as illustrated in Section 4.3. With this model, the system can automatically figure out that a light is not only a device, but also carries a potential conflict trigger attribute *brightness* in its connected area.

Together with the environmental information derived by *Preprocessor*, the integrated knowledge base makes the *Conflict Detector* more intelligent in detection of strong or weak conflicts, and further classification of C1, C2 and C3 weak conflicts.

(3) *Conflict Solver.* This component takes detected conflicts derived from *Conflict Detector* as input, and applies the heuristic conflict solution policy to solve the conflicts.

We construct a knowledge base for the *seriousness indicators*, as illustrated in Section 5.1. All of the devices capabilities involved in any type of conflicts can be attached to their seriousness level. With the *seriousness indicator*, the system can automatically figure out that an alarm is a device carrying a potential conflict trigger attribute soundness in its connected area with medium level of seriousness. As mentioned in the previous section, the *conflict frequency* is calculated through simple linear algebra on the temporal conditions included in the preconditions of transitions. In cases where our tool cannot figure out the frequency of a conflict, the conflict is shown as a "frequency unknown" group.

Along with the *seriousness indicators* and *conflict frequency*, *Conflict Solver* automatically provides detailed information on the conflicts and the well-designed solutions with user interaction involved.

(4) *Emulator.* The fourth component is an emulator that offers a virtual environment for executing the post-preprocessed SmartApps. The emulator uses a GUI to display the status of devices and accepts outside stimulations. It uses multithreading and event listeners to materialize the asynchronous and even-driven nature of SmartApps. The emulator is not necessary for the conflict detection as we use static analysis to detect conflicts. However, it does offer a convenient way to collect execution data for verifications.

DIAC is developed as a tool to analyze, detect and solve inter-app conflicts which tackles the challenge to implement, emulate, and circumvent all the functionalities enabled by the SmartThings platform. The implementation of DIAC integrates all the proposed techniques and the needed communications and the interactions with the physical devices.

## 7 EVALUATION

We evaluate DIAC using more apps than those in our previous work [19] and collect 113 public SmartApps from SmartThings Github repository; they are chosen to cover most of the devices and capabilities. The selected apps are detected based on the refined conflict definition and the detected conflicts are given with corresponding solution with user interaction involved.

Since conflict solution is provided in none of the previous methods, DepSys [23], SIFT [21] and HomeOS [9], we only compare the detection results by DIAC on 113 selected SmartApps with them.

We additionally conducted a user study. Twenty users filled a questionnaire. Seven computer science graduate students, three in other majors, and ten other professionals. Eight had prior experience with certain SmartHome systems. They were all given some background knowledge on SmartThings before the experiments. The users were given the 113 SmartApps and the descriptions of their controls of devices. The questionnaire consisted of 130 questions on those apps, with each corresponding to a conflict detected by any of the those previous or our method. To avoid biases, the users were not given any specific definitions of inter-app conflicts. The users were asked to mark those they felt as true conflicts affecting their expected behaviors of some of the apps and preferred solutions on each conflict. Solutions could be chosen from uninstalling an app, disabling an app, or ignoring the conflict. They in addition were asked to write down any other conflicts. From these results, we put together a collection of conflicts and the corresponding solution marked by any of the users. It is worth noting that 78% or more of the users agree on over 90% of the questions. We call the collection *users' collection*.

### 7.1 Results from DIAC

DIAC overall detects 27 strong conflicts and 46 weak conflicts. Our manual examination of the source code verifies that these detected conflicts are all of the right type according to our definitions of strong and weak conflicts. Table 4 shows examples of DIAC detected strong and weak conflicts, followed by solution report.

Our method makes conservative assumptions on unknown variables, which could potentially cause false alarms. But the influence is minor, for the simplicity of actual IoT apps. As lightweight device controllers, IoT apps are typically simple, with only 60–400 lines of source code in each, and a big portion of the code is meta info rather than controlling code. Our tool DIAC detects all the conflicts in *users' collection*. However it marks extra conflicts, which are regarded as false conflicts by most users. These cases are C2 or C3 weak conflicts. One of the reasons is that our device modeling overestimates the influence of some devices. An example is the interplay between `MachineChecker` and `MedicineReminder`. `MachineChecker` sounds an DingDing alarm when the vibration of a washing machine is sensed, while `MedicineReminder` plays a medicine reminder through a music player. Our tool regards them conflicting because the music player and the alarm have a C2 weak conflict through a common influence zone. DIAC then forces

MedicineReminder to be disabled as the seriousness level of an alarm is higher than that of a music player. However, no user marked them a conflict.

The seriousness of a weak conflict relates with the involved devices, and is sometimes subject to user's expectations. A detailed investigation of *users' collection* discovers another example similar to conflicts between MachineChecker and MedicineReminder. In SoulRelax, a music player plays music when it detects a writer working at a computer for a certain time length to let the user's brain rest. COMonitor sound an alarm when it senses the exceeded concentration of CO in the kitchen. The sound of the music player may drown out the alarm of the CO sensor. The two capabilities both affect the soundness attribute in the area, which allows our inference to detect the weak C2 conflict between apps. Interestingly, no user marks this as a false conflict. We suggest users to change the default solution (given by the heuristic policy) on the detected weak conflict if they prefer other options (e.g., disabling an app in certain scenario or ignoring the conflict).

We next use several typical examples to provide more detailed discussions on the conflicts detected and their corresponding solution by DIAC.

*Case 1:* Device bulb has two capabilities: ColorControl and ColorTemperature. In LightTheHall, the bulb in the hall way can be set to different colors when contact opens or motion is active. ButtonPush sets the bulb to a specific color temperature when a button is pushed. The two capabilities, color selection and temperature setting, both affect the color of the light. The device models hence help the constructor to put an influence edge between the two vertices, which allows our inference to detect the weak conflict between the apps. Since the level of the seriousness indicator of devices in LightTheHall and ButtonPush are the same, the different controls could be both wanted by a user. Based on the heuristic conflict solution policy, DIAC requires further decisions from users by providing options: disabling any one of the apps or ignoring the conflict.

*Case 2:* SmartCleaner randomly sets a robot cleaner in cleaning mode to clean the living room. In MovingDisabled, the Auto-WheelChair begins to move in the living room as it senses the moving intension of the handicapped person. There is a conflict between a robot cleaner and an auto-wheelchair moving with a handicapped person.This will produce a weak conflict when the robot cleaner is cleaning on the way of the auto-wheelchair.Through the device models and the models of influence zones, DIAC immediately detects this weak conflict falling into C3 category. According to the seriousness indicators of the robot cleaner and auto-wheelchair, DIAC then disables the execution of the robot cleaner to guarantee the safety of the handicapped person in the auto-wheelchair. The solution of SmartCleaner and MovingDisabled examples meets the choices from users in *users' collection*.

*Case 3:* CoDoor closes the door if any window opens. TempOn and ComfRoom are two green living apps, trying to save energy. TempOn turns off the thermostat when any window opens, and ComfRoom opens the door if the thermostat is off and closes the door if the thermostat is on. Our tool finds that a solution of CoDoor (window=1, door=0) cannot be implied by any solution of the conjunction of the apps. It hence claims the existence of a strong conflict among the three apps. This conflict can be intuitively understood. Consider the state when a window opens. It prompts TempOn to turn off the thermostat, which prompts ComfRoom to open the door. But opening the window also prompts CoDoor to close the door, forming a direct control conflict on the door. This is a strong conflict as TempOn and ComfRoom together disable the action of CoDoor. When two of these three apps are already installed, the installation of the third app will be rejected to avoid unintended changes on other apps. In our experiment, CoDoor and TempOn are installed before ComfRoom. The installation request of ComfRoom is rejected by DIAC.

Table 4. Examples of Conflicts Detection and Solution

| Apps in Conflict | Conflict Type | | | | Solution Report |
|---|---|---|---|---|---|
| | Strong | Weak(C1) | Weak(C2) | Weak(C3) | |
| NoticeBob<br>OccupiedLivingRoom | √ | | | | Info:Notice from light could get disabled in NoticeBob.<br>Installation of OccupiedLivingRoom is rejected. |
| OccupiedBedroom<br>WelcomeHome | √ | | | | Info:Turning on light in OccupiedBedroom could get disabled.<br>Installation of WelcomeHome is rejected. |
| CoDoor<br>TempOn<br>ComfRoom | √ | | | | Info:Closing the door in CoDoor could get disabled.<br>Installation of ComfRoom is rejected. |
| PowerSaving<br>TempOn | | √ | | | Info:Regulating the temperature in PowerSaving and TempOn is opposite.<br>Please choose your preferred action: Uninstalling TempOn, Disabling PowerSaving or TempOn (as default), Ignoring the conflict. |
| WetControl<br>AutoAd | | √ | | | Info:Humidity controlling in WetControl and AutoAd is opposite.<br>Please choose your preferred action: Uninstalling AutoAd, Disabling WetControl or AutoAd (as default), Ignoring the conflict. |
| CarTemp<br>SunnyWarm | | √ | | | Info:Regulating the temperature in CarTemp and SunnyWarm is opposite.<br>Please choose your preferred action: Uninstalling SunnyWarm, Disabling CarTemp or SunnyWarm (as default), Ignoring the conflict. |
| VibrationNotice<br>MedicineReminder | | | √ | | Info:Sound of siren in VibrationNotice and sound of music player in MedicineReminder are displayed.<br>Seriousness indicator of VibrationNotice and MedicineReminder are not the same. VibrationNotice will be executed. MedicineReminder will be disabled. |
| SMS-Reminder<br>CallMeWhen | | | √ | | Info:Phone message in SMS-Reminder and phone call in CallMeWhen are displayed.<br>Seriousness indicator of SMS-Reminder and CallMeWhen are the same. SMS-Reminder and CallMeWhen will be executed. Conflict is ignored. |
| SoulRelax<br>CoMonitor | | | √ | | Info:Sound of music player in SoulRelax and sound of alarm in CoMonitor are displayed.<br>Seriousness indicator of SoulRelax and CoMonitor are not the same. SoulRelax will be executed.CoMonitor will be disabled. |
| LightTheHall<br>ButtonPush | | | | √ | Info:The brightness could be controlled by both LightTheHall and ButtonPush.<br>Seriousness indicator of LightTheHall and ButtonPush are the same.<br>Please choose your preferred action: Disabling LightTheHall or ButtonPush (as default), Ignoring the conflict. |
| WarmWelcome<br>SafetyVacation | | | | √ | Info:The bright could be controlled by both WarmWelcome and SafetyVacation.<br>Seriousness indicator of WarmWelcome and SafetyVacation are the same.<br>Please choose your preferred action: Disabling WarmWelcome or SafetyVacation (as default), Ignoring the conflict. |
| MachineCall<br>WarmWelcome | | | | √ | Info:The temperature of color could be controlled by both MachineCall and WarmWelcome.<br>Seriousness indicator of MachineCall and WarmWelcome are the same.<br>Please choose your preferred action: Disabling MachineCall or WarmWelcome (as default), Ignoring the conflict. |
| SmartCleaner<br>MovingDisabled | | | | √ | Info:The space of room could be controlled by both SmartCleaner and MovingDisabled.<br>Seriousness indicator of SmartCleaner and MovingDisabled are not the same. SmartCleaner will be disabled.MovingDisabled will be executed. |

Table 5. Comparison Results

| | Recall | Precision |
|---|---|---|
| DIAC | 100% | 88% |
| Adapted DepSys [23] | 51% | 63% |
| SIFT [21] | 24% | 100% |
| HomeOS [9] | 60% | 75% |

### 7.2 Comparison with Previous Methods

We compare our detection recall and precision to those of adapted DepSys [23], SIFT [21] and HomeOS [9] on the 113 apps. Table 5 reports the results. DepSys requires users to specify the priorities of apps and the emphasis on different actions; it defines and detects conflicts based on these specifications. It is not applicable to SmartThings apps directly. We instead get the results based on adapted definition which resembles the conflicts definition in DepSys to a large degree (without the priorities and emphasis needed from user specifications). The results of SIFT and HomeOS are obtained by following the detection methods described in the previous papers.

The adapted DepSys method detects 15 out of 27 strong conflicts, 25 out of 52 true weak conflicts and gives 12 false conflicts. Twelve of the missed strong conflicts are similar to the `secHouse example`, in which the control conditions of one app is affected by others. One of the missing conflicts is `TempOn`, `ComfRoom`,`CoDoor`. The adapted DepSys misses 51% weak conflicts. Many of the missed weak conflicts involve controls of different devices that have same effects on the environment. Some of the false conflicts are due to the overestimate of the interplays between apps on users, while some are the cases where the apps have the same controls on the same devices. An example is `ProtectRoom` and `SafetyVacation` in a setting such that both try to turn off a light when no motion is detected for 2min. DepSys mistakenly labels them as a conflict because it sees that they may control the same device at the same time.

SIFT achieves the same performance in strong conflict detection as DepSys does. But because it considers only controls of common devices rather than the effects of devices, it finds only 9 weak conflicts. HomeOS has a higher recall than that of SIFT. It is because HomeOS labels any two apps accessing a common device as a conflict, while SIFT considers only incompatible actions on the same device. Two apps may both turn on a switch to signal the occurrences of different events. Their actions are the same (turning on the switch), but they form a weak conflict as they may create a confusion to the user on the meaning of the signal. SIFT misses the conflict but HomeOS captures it.

Our method outperforms previous methods for its more rigorous definitions of conflicts and its detection methods.

### 7.3 Overhead

We measured the overhead of the conflict detection and solution on two platforms: an Intel desktop equipped with a core-i5 CPU (2.5GHz) and an Android Smartphone equipped with an ARM Dual-core 1.2 GHz Cortex-A9. The conflict detection for an app against the other 112 apps takes less than 0.05s and 0.33s on the two machines, respectively.The conflict solution is given in a negligible time.

Most time in the conflict detection is spent on the execution of the detection algorithm; the construction of IA Graph and other operations take negligible time. IoT apps are typically small as mentioned earlier, and the IA Graph only captures code related with device capabilities. As a result, the constructed IA Graph is small, involving no more than 20 transitions each.

### 7.4 Multi-Platform Applicability

DIAC supports different IoT platforms by design. The derivation of IA Graphs by the *Conflict Detector* in DIAC is platform-specific since platforms may use different languages and customized APIs. Our work shows that *Conflict Detector* works well in representation of IoT controls and inter-app interplays from the source code of Groovy-based SmartApps. Therefore, the only engineering effort for supporting multiple platforms is to implement the derivation of IA Graphs for other languages. Other than programs/apps, some platforms define rules through templates. For example, IFTTT [14] provides graphical interfaces for users to define automation rules by selecting pre-defined templates and filling out parameters. IoT controls can be extracted by crawling text data on the related pages and parsing the texts with natural language processing technologies.

## 8 CONCLUSIONS

This paper has presented a systematic study on detecting and solving various conflicts among IoT apps in an open IoT platform. It gives a more precise characterization of inter-app conflicts than previous studies do. It provides formal definitions of inter-app strong conflicts and weak conflicts. It proposes a formalism named IA Graphs to represent the controls of devices by IoT apps. It develops some fundamental theorems and algorithms for detecting inter-app conflicts upon IA Graphs. It designs a heuristic conflict solution policy combining seriousness indicator and conflict frequency for solving various detected conflicts. It incorporates all these techniques into an open-source system named DIAC for analyzing SmartThings apps. Experiments show that the techniques are effective in detecting and solving various inter-app conflicts. On 113 public SmartThings apps, it successfully detects 27 strong conflicts and 46 weak conflicts with no false alarms.

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1525609, CNS-1717425, CCF-1703487, the Fundamental Research Funds for the Central Universities under Grant No. 300102249105 and the Natural Science Basic Research Plan in Shaanxi Province of China under Grant No. 2019JQ-580. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, Ministry of Education of China and Natural Science Basic Research Plan in Shaanxi Province of China.

## REFERENCES

[1] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2012. Efficient system-enforced deterministic parallelism. *Commun. ACM* 55, 5 (2012), 111–119.

[2] Gérard Berry. 2000. *The Esterel v5 language primer: version v5_91.* Centre de mathématiques appliquées, Ecole des mines and INRIA.

[3] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable race detection for android applications. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 332–348.

[4] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 285–300.

[5] N. Cardozo, S. González, K. Mens, R. Van Der Straeten, and T. DHondt. 2013. Modeling and Analyzing Self-Adaptive Systems with Context Petri Nets. In *Proceedings of International Symposium on Theoretical Aspects of Software Engineering*.

[6] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2018. Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling. (2018).

[7] 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

[8] Robert I Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)* 43, 4 (2011), 35.

[9] C. Dixon, R. Mahajan, S. Agarwal, A.J. Brush, B. Lee, S. Saroiu, and P. Bahl. 2012. An Operating System for the Home. In *the USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*.

[10] Gartner, Inc. 2017. The world's leading information technology research and advisory company. https://www.gartner.com/technology/home.jsp.

[11] Pilar Gonzalez-Ferez and Angelos Bilas. 2014. Tyche: An efficient Ethernet-based protocol for converged networked storage. In *Symposium on Mass Storage Systems and Technologies*.

[12] Groovy 2017. Groovy Programming Language. http://www.groovy-lang.org.

[13] C.H. Hsiao, J. Yu, S. Narayanasamy, Z Kong, C.L. Pereira, G. A. Pokam, P. M. Chen, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Systems. In *International Conference on Programming Language Design and Implementation (PLDI)*.

[14] IFTTT 2017. IFTTT website. http://ifttt.com.

[15] ITU. 2015. Internet of Things Global Standards Initiative, Retrieved 26 June 2015.

[16] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. (2017).

[17] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 732–741.

[18] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 15–27.

[19] Xinyi Li, Lei Zhang, and Xipeng Shen. 2019. IA-graph based inter-app conflicts detection in open IoT systems. (2019), 135–147.

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

[21] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: Building an Internet of Safe Things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN '15)*. ACM, New York, NY, USA, 298–309. https://doi.org/10.1145/2737095.2737115

[22] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 316–325.

[23] Sirajum Munir and John A. Stankovic. 2014. DepSys: Dependency Aware Integration of Cyber-Physical Systems for Smart Homes. In *ICCPS '14: ACM/IEEE 5th International Conference on Cyber-Physical Systems (with CPS Week 2014) (ICCPS '14)*. IEEE Computer Society, Washington, DC, USA, 127–138. https://doi.org/10.1109/ICCPS.2014.6843717

[24] Wolfgang Reisig. 1985. Petri nets: an introduction, volume 4 of EATCS monographs on theoretical computer science.

[25] SmartApp 2017. SmartApp Repository. https://github.com/SmartThingsCommunity/SmartThingsPublic.

[26] SmartThings 2017. SmartThings by Samsung. http://www.smartthings.com.

[27] Yuan Tian, Nan Zhang, Yuehhsun Lin, Xiaofeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. In *USENIX Security Symposium*. 361–378.

[28] VisionMobile 2017. A leading analyst firm in the developer economy. https://www.visionmobile.com.

[29] Z3 2017. Z3 Theorem Prover by Microsoft Research. https://z3.codeplex.com.