IA-Graph Based Inter-App Conflicts Detection in Open IoT Systems

Xinyi Li*
Chang'an University
China
xinyili@outlook.com

Lei Zhang
North Carolina State University
USA
lzhang45@ncsu.edu

Xipeng Shen
North Carolina State University
USA
xshen5@ncsu.edu

Abstract

This paper tackles the problem of detecting 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 employs a graph representation (named *IA Graph*) for formally representing IoT controls and inter-app interplays. It provides an efficient conflicts detection algorithm implemented on a SmartThings compiler and shows significantly improved efficacy over prior solutions.

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

Keywords IOT, Compiler, Conflicts Detection

ACM Reference Format:
Xinyi Li, Lei Zhang, and Xipeng Shen. 2019. IA-Graph Based Inter-App Conflicts Detection in Open IoT Systems. In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3316482.3326350

1 Introduction

Recent years have witnessed a rapid development of Internet of Things (IoT) technology. In this work, we concentrate on *open IoT platforms*. Unlike closed IoT platforms (e.g., an IoT system for factory control), such platforms allow the public to develop and upload IoT apps for users to download and deploy in their IoT environments. In such platforms, the set of apps deployed in an IoT environment is not predefined, and these apps 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., SmartThings from Samsung [18], HomeOS [5]). On SmartThings, for instance, any developer may write an app in SmartThings SDK, and upload the app to SmartThings cloud. Any user can then download the app from the cloud, install and deploy it in her SmartThings environment (e.g., a house equipped with SmartThings hubs).

The openness however also brings issues. An important one is conflicts among apps. In an open IoT environment, usually many apps are deployed at the same time, interacting with the set of sensors and devices in the environment. Because these apps are independently developed, unexpected interplays among them often happen, resulting in undesirable consequences. An example includes 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, when the house is empty, the presence of people in a house by making the light stay on and off alternatively periodically (half an hour for each state). When these two apps are installed in the same environment, `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 [17] shows that such inter-app conflicts are common: Among 22 popular public SmartApps, we found 18 conflicting groups of apps (Section 5). The number of smart connected homes could rise from current 100 million to 700 million by 2020 [6], and about 1.5 million IoT developers are currently working on Smart Home projects [19]. With more sensors used and more apps developed, the inter-app conflict problem will become even more common.

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 [1]. The method does not apply to open IoT platforms as users may install arbitrary apps into them.

*The work was done when the first author was working at North Carolina State University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LCTES '19, June 23, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6724-0/19/06...$15.00
https://doi.org/10.1145/3316482.3326350

Recent years have seen several efforts on addressing inter-app conflicts on open IoT platforms, but they have all considered some basic types of conflicts: two apps access the same device (e.g., in HomeOS [5]) or have different/opposite effects on the same device (e.g., in SIFT [12]) or environment (e.g., DepSys [13]).

In this work, we argue that the previous definitions of inter-app conflicts are over simplified, and subsequently, the conflict detection methods that they have developed on the definitions are inadequate. We provide an improved definition that better aligns with users experience and intuition (validated through a user study), and categorize the conflicts into strong conflicts, weak conflicts, and implicit conflicts depending on their different natures (Section 3).

The rectified definition offers a more comprehensive coverage of conflicts, and correspondingly calls for more sophisticated methods for detecting these conflicts. Special challenges exist on two aspects: how to represent apps in a form amenable for detecting the more comprehensive range of conflicts, and how to effectively reason about inter-app relations to find all the conflicts and identify their types and seriousness.

The second part of this paper presents our solution to the challenges. It proposes a graph structure named *IA Graph* to concisely represent the controls in each IoT app and the various schedules of events (Section 4.1). Based on the representation, it uses an efficient algorithm to leverage first-order logic and SMT solvers to detect conflicts of all types. It employs a novel method to further distinguish weak conflicts of various levels of seriousness based on device modeling and conflict frequencies (Section 4.3).

We integrate the techniques together into a tool named DIAC (for Detector of IoT App Conflicts) and evaluate it on the SmartThings platform. The experiments demonstrate that DIAC can successfully detect the conflicts among those apps with marginal time overhead. Compared to results from previous work, DIAC shows significantly improved precision (90%) and recall (100%).

Overall, this work makes the following major contributions:

- It provides a systematic categorization of various inter-app conflicts in IoT and offers a new set of conflicts definitions that address some major issues in prior definitions.
- It designs an *IA Graph* to concisely represent the controls in each IoT app and the various schedules of events.
- It provides an algorithm and a compiler-based implementation for automatically detecting various inter-app conflicts, and identifying their types and seriousness.

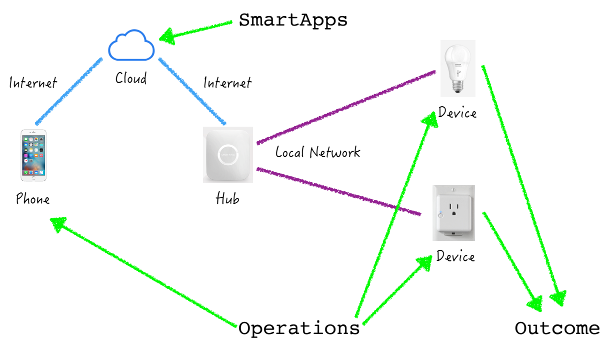

Figure 1. SmartThings Architecture

2 Premises and Background

Focus The problem focused in this work is inter-app control conflicts. Debugging within an app [3, 11] is outside the scope of this work.

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 [9], in which, app developers just write a few high-level rules. The latter is represented by Samsung SmartThings [18], in which, app developers write apps in an SDK on scripting language `Groovy` [7]. 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. With SmartThings SDK, everyone can write and publish SmartApps on SmartThings cloud; it is free for users to download SmartApps from the cloud. Once a SmartApp is installed on 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 [7], 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. In our example, this part contains only one section "Select devices". 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

```
//Metadata of the app
definition(
  name: "A Sample SmartApp",
  description: "Turn a light on when …"
…)

//Options used at installation time.
preferences{
 section("Select devices"){
 input "contact1", "capability.contactSensor", \
                      title: "Select contact sensor"
 input "light1", "capability.switch", \
                      title: "Select a light"   } }
```

```
//Pre-defined methods called
// at installation and updates.
def installed(){ initialize()}
def updated(){
  unsubscribe()
 initialize() }

def initialize(){
  subscribe(contact1,
 "contact.open",   openHandler)}

//Event handler functions
def openHandler(evt){
  light1.on() }
```

Figure 2. A SmartApp.

"contactSensor" capability in the user's SmartThings environment, and ask the user to pick the one she 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". Method "openHandler" is defined at the end of the example, which 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.

3 Definitions 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.

3.1 Prior Definitions

Three definitions have been used in previous studies on detecting inter-app conflicts.

(1) Definition 1: Two apps conflict if they access the same device at the same time (used in HomeOS [5]).

(2) Definition 2: Two apps conflict if they try to cause different (and incompatible) actions on the same device simultaneously. This definition is used in SIFT [12].

(3) Definition 3: 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 [13] uses.

The three definitions form an evolving path, getting increasingly more sophisticated. For instance, the first definition is quite rudimentary and would even inappropriately regard readings of the same sensor by two apps as a conflict. Definition two avoids that issue, while definition three 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. Below is an example.

> *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.

3.2 Our Definitions and Categorizations

Our new definition 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.

We note that different controls of a device as captured by *weak conflicts* are not always unacceptable to a user. Section 4.3 will discuss different seriousness of *weak conflicts* and describe how the detection algorithm refines *weak conflicts* through device modeling and conflict frequency.

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. Implicit conflicts may also happen on different capabilities of a single device. For instance, one security app takes a 5-minute video whenever a motion is detected during some period of time, while another app takes a picture every 10 minute. Video taking and picture taking are different capabilities of the camera, however, both use the lens of the camera; when one is happening, the other is disabled. 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.

Conflicts and Resolutions 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. The categorization helps us design detection algorithms suitable for each type of the conflict. It can also potentially help with discriminative resolutions of different types of conflicts.

4 Conflicts Detection

When the definition of conflicts is narrow, the detection can be simple. For instance, for the conflicts defined in Definition 1, the detector needs to check only the set of devices each app accesses as HomeOS does [5]. 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.

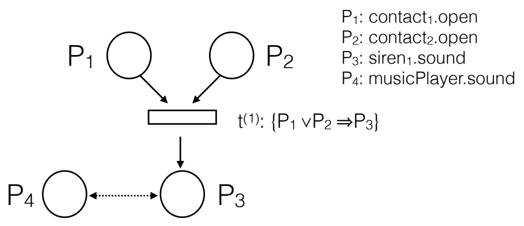

Figure 3. A simple IA Graph.

Our designed conflict detection method consists of two key components. To assist the reasoning, we design a graph representation to concisely 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 $\implies$ 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*.
- E_i: a set of *influence edges*. Each influence edge is a bidirectional 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 [7], 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 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".

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*

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 [2, 16]. 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$\leq$ contact.open.chronos$\leq$ 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` $\Longrightarrow$ `light=1` in app1, `t2: switch=1` $\Longrightarrow$ `light=0` in app2. Function F would be $(light \vee -switch) \wedge (-light \vee -switch)$. First-order logic simplifies it into $-switch$. 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

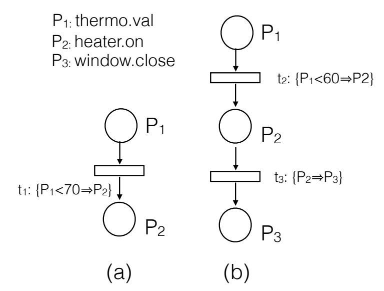

Figure 4. Two apps with weak conflicts on P_2.

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 between conflict seriousness and device modeling and conflict frequency. Specifically, we categorize device capabilities into the following three classes:

- (C1) Device capabilities for exclusive control. For these capabilities, two different controls usually create unfavorable interferences. Examples include heater.on, cooler.on, hydrator.on, dehydrator.on.
- (C2) Device capabilities for shared control. These capabilities are usually intended for multiple uses; different controls on them (if they do not disable one another) are often acceptable. Besides sirent.sound, other examples include camera.shot, 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 of such capabilities include lock.open, stereo.play.

To further distinguish C3 weak conflicts of different seriousness, we add *conflict frequency* into consideration. First, we introduce a term, 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 a device capability is the smaller value of the frequencies of the two controls. For the SafeHouse app, the frequency of its control of light.on is 24; the frequency in ESave is 720. The conflict frequency of them is 24. The conflict frequency of sunsetApp and earlyRiseApp is 1. Our detection algorithm heuristically rates more frequent weak conflicts more serious. 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 C3 conflicts are shown as a "seriousness unknown" group.

To our knowledge, this is the first proposed solution that distinguishes weak conflicts of different seriousness. Its effectiveness is validated through a user study in Section 5.

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. On the other hand, picture taking and video taking by a camera both affect the availability of the lens of the camera. They form strong conflicts as one would disable the firing of the other at a given moment. The seriousness of an implicit weak conflict is determined in a way similar to normal weak conflicts, except that if the involved capabilities belong to different categories (C1,C2,C3), the more serious category decides the category of the conflict.

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. Figure 5 outlines the algorithm.

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
camera.picTaking	lens	availability	strong
camera.videoTaking	lens	availability	strong

```
Input: A: the existing set of apps;
       b: a new app to install;
Output:
 SC: strong conflicts;
 WC: weak conflicts;
 SCI: strong conflicts due to influence zones;
 WCI: weak conflicts due to influence zones;

Algorithm:
 // P: set of places that b shares with A
 // InfSets[p]: the set of places of A that overlap with p in terms of
              influence zones, where, p is controlled by b

 G = AddToIAGraph(b, A.AIGraph);
 findSharedPlaces (G, b, &P, &InfSets);
 foreach p in P
   p.f[b] = getControlFormula(G, b, p)
   // detect weak conflicts on the same capability
   foreach a in A
     if (p.f[a] $\not\Leftrightarrow$ p.f[b])
       WC.add(<p, a>, calSeriousness(p,a,b));
   end for
   // detect strong conflicts on the same capability
   p.f* = conjunction of all p.f[i]   ($i \in A \cup b$);
   for i in  $A \cup b$
      if (!compatible(p.f*-p.f[i], p.f[i]))
        SC.add(<p, i>);
   end for
 end for
// detecting influence zone caused conflicts
foreach s in InfSets
  if (s consists of places from multiple apps)
    if (s.InfType == WEAK)
      WCI.add(s, calSeriousness(s));
    else if (s.InfType == STRONG)
      r = conjunction of f* of all the places in s
      if ($\exists$ p, q in s,  p!=q & app(p)!=app(q) & r $\Rightarrow$ p^q )
        SCI.add(s);
end for
```

Figure 5. Inter-App Conflict Detection Algorithm.

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. The seriousness of the conflict is calculated based on the description in Section 4.3. A record is added into the weak conflict set WC. 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 procedure "compatible(f_1, f_2)" in strong conflict detection is worth more explanations. It 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 [20]) 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.

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.

4.4 Discussions on the Impact to Conflict Resolution

The focus of this work is on conflict detection; how to resolve conflicts is beyond our scope. This part mainly discusses the implications of the conflict detection to conflict resolution.

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 resolution ultimately depends on the user and the specific context and scenario. Since different users have different intentions and preferences 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 resolution 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 and their levels of seriousness. Thanks to the design of its detection algorithm, it can further provide more detailed information on the conflicts: For a strong conflict, the answers by the SMT solver can give the information on which functionality of which app could get disabled in what scenarios; for a weak conflict, the detection can give the information on which apps have different controls on which device capabilities. These information could potentially help users make better decisions when selecting the options to resolve the conflicts (e.g., disabling an app in certain scenario, uninstalling an app, or ignoring some conflicts).

c-1. App19:

```
def installed(){
  subscribe (contactSensor, \
       “contact.open”, eventHandler)
  subscribe (motionSensor,  \
       "motion.active", eventHandler)
}
def eventHandler (evt) {
  def hueColor = 0
  def saturation = 100
  def lightLevel = 100
  switch(color) {
    case "Daylight":
      hueColor = 53
      saturation = 91
      break
    case "Soft White":
      ...}
 def newValue = [hue: hueColor, \
    saturation: saturation, \
    level: lightLevel]
  bulb.setColor(newValue)
}
```

c-2. App20:

```
def installed() {
  subscribe (button, “button”, \
    changeColorTemp)
}
def changeColorTemp(evt) {
  if (evt.value == “pushed”) {
    def temp = colorTemperature \
      as Integer ?: 4100 \
      bulb.setColorTemperature(temp)
  }
}
```

Figure 6. Example Apps.

5 Evaluation

We implemented the proposed techniques and name the resulting framework *DIAC* (for Detector of IoT App Conflicts).

Table 3. Inter-app Conflict Detection Results

App index	Description of SmartApp	Device Involved	Number of Conflicts	
			Strong	Weak
1	Turn on a switch when CO2 levels are too high.	Co2Sensor,switch1		1(6)
2	Lock a door at a specific time if it is closed.	contactSensor1,lock	1(11)	2(11,13)
3	Randomly turn on/off lights to simulate the appearance of a occupied home.	presenceSensor1, presenceSensor2, presenceSensor3, light1, light2, light3	5(4, 5, 13, 18, 11)	
4	Turn on the hall light if someone comes home and the door opens.	presenceSensor1, contactSensor1, light1, light4	1(3)	4(5*,7*,13,18)
5	Turn your lights off after a period of no motion being observed.	motionSensor1, light1, light2, light3	2(3,13)	4(4*,7*,13,18)
6	Turn on Monitor and sound alarm when vibration is sensed.	accelerationSensor1, switch1, alarm		3(1,9*,16*)
7	Turns on a light when vibration sensed.	accelerationSensor2,light4		3(4*, 5*,13,18)
8	Turns off device when wattage drops below a set level after a set time.	powerMeter,thermostat1		2(10*,11)
9	A reminder of taking medicine (determined by whether a drawer has been opened) at a specified time.	contactSensor2,musicPlayer		2(6*,16*)
10	Turn off the thermostat when any window or door opens.	window1, window2, thermostat2, door	2(21,22)	2(8*,11*)
11	Turns on/open selected device(s) at a set time on selected days of the week only if a selected person is present and turns off/close selected device(s) after a set time or receive a light signal.	presenceSensor1, thermosta, t1, thermostat2, lock,garageDoor, light2, switch2, bulb	3(2,3,15)	9(2, 8, 10*, 13, 15*, 16, 17, 19*, 20*)
12	Take a burst of photos when contact opens, vibration detected, motion or presence sensed or switch turns on.	contactSensor1, acceleration, Sensor1, motionSensor1, presenceSensor1, switch2, camera1, camera2	1(15*)	
13	Select locks, presence, motion, acceleration or contact sensors to control a set of lights.	motionSensor1, motionSensor2, motionSensor3, presenceSensor1, presenceSensor2, accelerationSensor1, accelerationSensor2, contactSensor1, contactSensor2, lock, light1, light2, light3, light4	2(3,5)	9(2, 4, 5, 7, 11, 15*, 18, 19*, 20*)
14	Close a selected valve if moisture is detected.	waterSensor,valve		
15	When door opens, change the bulb color and take a video as a safeguard.	contactSensor2,bulb,camera2	2(11,12*)	4(11*, 13*, 19*, 20*)
16	Turn on switch at someone's presence.	presenceSensor2,switch2		4(6*,9*,11,17)
17	Turn on the warning switch when smoke is detected.	smokeDetector,switch2		2(11,16)
18	Turn lights off when no motion and presence is detected for a period of time.	motionSensor1, motionSensor3, presenceSensor1, presenceSensor2, light1, light4	1(3)	4(4*,5*,7,13)
19	Sets the colors and brightness level of lights to match mood when contact opens or motion is detected.	contactSensor2, motionSensor2, bulb		4(11*, 13*, 15*, 20*)
20	Sets your bulb to default Moonlight (4100) when a button is pushed.	button,bulb		4(11*, 13*, 15*, 19*)
21	Close the door if any window opens.	door,window1,window2	2(10,22)	1(22)
22	Open the door if thermostat is off and close the door if thermostat is on.	door,thermostat1	2(10,21)	1(21)

This section first reports the results of DIAC on 22 SmartApps, then compares with those from previous methods, and reports the runtime overhead of our technique.

Methodology We collected 22 public SmartApps from SmartThings Github repository; they are chosen to cover a good range of devices and capabilities. Together, they cover 33 devices of 21 types and 22 capabilities. Figure **??** (a) shows the description of each SmartApp and the involved devices.

We compare our results with those by previous methods, DepSys [13], SIFT [12] and HomeOS [5].

We additionally conduct a user study. Twenty users filled a questionnaire. Seven computer science graduate students,

Table 4. Comparisons

	Recall	Precision
our DIAC	100%	90%
Adapted DepSys [13] (Def3)	53%	74%
SIFT [12] (Def2)	29%	100%
HomeOS [5] (Def1)	63%	84%

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 22 SmartApps and the descriptions of their controls of devices. The questionnaire consists of 49 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 feel as true conflicts affecting their expected behaviors of some of the apps. They in addition were asked to write down any other conflicts. They suggested none. From these results, we put together a collection of conflicts marked by any of the users; these conflicts are regarded as the conflicts that a conflict detector should detect, as they affect some of the users' experience. It is worth noting that 75% or more of the users agree on over 90% of the questions. We call the collection *users' collection*. Eleven out of the 49 cases were rejected by all users.

5.1 Results from Our Method DIAC

The fourth and fifth columns in Figure **??** (a) report the detailed numbers of detected strong and weak conflicts by DIAC, where, in the parentheses we mark the ID numbers of the apps that conflict with the app on that row. We use * to indicate the conflicts caused by influence zones. Our tool overall detects 8 pairwise strong conflicts and 11 groups of weak conflicts in Figure **??**. Each weak conflict group contains 2 or 3 apps. 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.

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 four extra conflicts, which are regarded as false conflicts by all users. One of the reasons is that our device modeling overestimate the influence of some device. An example is the interplay between App 6 and App 9. App 6 sounds an alarm when vibration is sensed, while App 9 plays a medicine reminder through a music player. Our tool regards them conflicting because the music player and the alarm have an implicit conflict through a common influence zone. However, no user marked them a conflict. The other cases are C2 or non-serious C3 weak conflicts.

The seriousness levels suggested by DIAC show a strong correlation with the percentages of users marking a conflict. All strong conflicts and C1 weak conflicts have a percentage over 90%, all C2 weak conflicts have a percentage less than 20%. The seriousness levels of all conflicts (except five "seriousness unknown") have a significant Spearman's rank correlation coefficient (0.8) with user percentages.

We next use several case studies to provide more detailed discussions on the conflicts detected by DIAC, and then give the comparisons with the results from previous methods.

Case 1: Conflicts on different devices. This case involves three apps. App21 closes the door if any window opens. App10 and App22 are two green living apps, trying to save energy. App10 turns off the thermostat when any window opens, and App22 opens the door if the thermostat is off and closes the door if the thermostat is on.

Our tool finds that a solution of App21 (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 App10 to turn off the thermostat, which prompts App22 to open the door. But opening the window also prompts App21 to close the door, forming a direct control conflict on the door. This is a strong conflict as App10 and App22 together disable the action of App21.

Case 2: Implicit strong conflicts on the same device. This is a conflict between video taking and picture taking by cameras. App12 takes a burst of photos when acceleration is active, motion is active, people present, or the switch is on. In App15, video recording starts when contactSensor2 is open. When recording videos, a camera cannot take pictures for another app. Therefore, if motionSensor1 is active after contactSensor2 opens and the video recording lasts 1 minute, the image of the active motion cannot be captured. The combined IA Graph of the two apps have an influence edge connecting the two capabilities. Through the device models shown in Table 2, DIAC immediately detects the strong conflicts.

Case 3: Implicit weak conflicts on the same device. Device bulb has two capabilities: ColorControl and ColorTemperature. Figure 6 (c-1) shows App19, in which the bulb can be set to different colors when contact opens or motion is active. App20 in Figure 6 (c-2) 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 put an influence edge between the two vertices, which allows our inference to detect the weak conflict between the apps.

5.2 Comparison with Previous Methods

We compare our detection recall and precision to those of adapted DepSys [13], SIFT [12] and HomeOS [5] on the 22 apps. Figure **??** (b) 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 Definition 3 given in Section 3.1, 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 6 out of 9 strong conflicts, 14 out of 29 true weak conflicts and gives 7 false conflicts. Two of the missed strong conflicts are similar to the `secHouse example`, in which the control conditions of one app is affected by others. They are the conflicts among App2, App11, and App15, and the conflicts among App10, App21, App22. The third missed strong conflict is about camera usage between App12 and App15. The adapted DepSys misses 52% weak conflicts. Many of them 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 App5 and App18 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 the DepSys does. But because it considers only controls of common devices rather than the effects of devices, it finds only 6 weak conflicts. HomeOS has a 63% recall, higher 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 but 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.

5.3 Overhead

We measured the overhead of the conflict detection 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 21 apps takes less than 0.03s and 0.2s on the machines respectively.

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 capture code related with device capabilities. As a result, the constructed IA Graph are small, involving no more than 20 transitions each.

Recall that the conflict detection 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 ith common vertex. In our collection of the 22 apps, $|P|$ equals 16, and $n(i)$ is no more than 7 (contactSensor.open is the most popular vertex). The longest formula involves 10 device capabilities while most contains much less than that. So the detection finishes quickly.

6 Related Work

Prior sections have already discussed the relations between this work and the previous conflicts detections in open IoT. We hence skip them in this section. One other work worth mentioning is the IOT calculus published recently [14]. The work presents a formal calculus for IOT. It is complementary to this work in the sense that the more clearly captured semantics by the calculus could potentially help refine the implementations of our detection algorithms. The evaluation part of that work includes conflict detection as one of the possible uses of the calculus. As that part is only a case study, the work does not give deep discussions on the conflict definitions. The definition of conflicts it uses is the same as the previous definition in SIFT [12] (i.e., Definition 2 mentioned in Section 3.1), which has already been compared with in the previous section.

In a broad sense, our use of IA Graphs to model the apps and the use of Horn Clause logic to detect conflicts could be regarded as a variant of the model checking approach. Model checking is a method widely used for verifications of programs and (concurrent) reactive systems. The related work goes back to 1980s [15] and earlier. The key contribution of our work is that it, for the first time, systematically explores formal modeling of inter-app conflicts in open IoT environments, making the problem approachable by a customized model checking method. More specifically, this work features the following: (1) *Conflict definitions.* We give new definitions of inter-app conflicts in open IoT environments, which essentially define the models for checking; (2) *Representations.* We design IA Graphs to formally represent controls in IoT apps and capture their interplays; (3) *Influence zone.* We introduce the concept of influence zone and implicit conflicts, which are special for modeling IoT conflicts comprehensively; (4) *IoT device and API modeling.* These models help generate formal representations that can well capture interplays of device controls coded in the IoT programming language.

There are some recent studies on debugging the correctness issues of IoT control systems [3, 11]. 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.

Conflicts could happen in traditional real-time/reactive systems [4, 10]. 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 cooperatively. The development of these systems often start with some formal specifications of the design, 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 inter-app conflicts.

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 [8]. They concentrate on data races, rather than inter-app conflicts.

7 Conclusions

This paper presents a new set of definitions and categorizations of inter-app conflicts in open IoT systems, and employs *IA Graph* for formally representing IoT controls and inter-app interplays. It provides an efficient conflicts detection algorithm, which shows a significantly better efficacy than prior methods.

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

[2] 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*.

[3] 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

[4] 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.

[5] 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)*.

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

[7] Groovy 2017. Groovy Programming Language. (2017). http://www.groovy-lang.org.

[8] 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)*.

[9] IFTTT 2017. IFTTT website. (2017). http://ifttt.com.

[10] 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.

[11] 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

[12] 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

[13] 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

[14] J. Newcomb, S. Chandra, J. Jeannin, C. Schlesinger, and M. Sridharan. 2017. IOTA: A Calculus for Internet of Things Automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 119–133. https://doi.org/10.1145/3133850.3133860

[15] J.P. Quielle and J. Sifakis. 1981. Specification and verification of concurrent systems in CESAR. In *Proc. 5th International Symposium on Programming*.

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

[17] SmartApp 2017. SmartApp Repository. (2017). https://github.com/SmartThingsCommunity/SmartThingsPublic.

[18] SmartThings 2017. SmartThings by Samsung. (2017). http://www.smartthings.com.

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

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