A Systematic Examination of Inter-App Conflicts Detections in Open IoT Systems

Xinyi Li
Computer Science and Technology Department, Xi’an JiaoTong University & NCSU

Lei Zhang
Computer Science Department, NCSU

Xipeng Shen
Computer Science Department, NCSU

Yong Qi
Computer Science and Technology Department, Xi’an JiaoTong University

Abstract

Recent several years have witnessed the rapid development of open platforms for developing and deploying Internet of Things (IoT) apps. One of the important issues in such environments is the unintended interactions among independently developed apps installed in a single environment. Although there have been some recent studies trying to address the issue, the treatments to the problem remain preliminary, reflected by the rudimentary definitions of inter-app conflicts and the entailed inadequacy of the previously proposed solutions.

This paper strives to provide a deeper and more systematic investigation to the problem. It offers a series of new definitions and categorizations to more precisely characterize the nature of IoT inter-app conflicts, and proposes a new representation named IA Nets for representing IoT controls and inter-app interplays. It then describes an efficient conflicts detection algorithm developed upon these new definitions and representations. It further presents DIAmond, a compiler and runtime framework that integrates all the proposed techniques together into a comprehensive solution. Experiments on SmartThings apps validate its effectiveness for IoT inter-app conflicts detections.

1. Introduction

Recent years have witnessed a rapid development of Internet of Things (IoT) technology. Generally speaking, IoT is the internetworking of physical devices, vehicles, buildings and other items-embedded with electronics, software, sensors, actuators, and network connectivity that enable these objects to collect and exchange data (ITU, 2015). Many existing IoT platforms are industrial special-purpose platforms. Examples include IoT systems for factory controls, systems for medical service in a hospital, and so on. We call such IoT platforms closed IoT platforms, in which, the whole system is designed coherently with a central plan and the set of apps and operations are usually predefined, conforming to some model created beforehand by a group of expert developers.

In this work, we concentrate on open IoT platforms. Unlike closed IoT platforms, 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.

Thanks to the superior extensibility and accessibility provided by open IoT platforms, such platforms are becoming increasingly influential, exemplified by the fast development of home automation systems (e.g., SmartThings from Samsung (SmartThings), HomeOS (Dixon et al., 2012)). On SmartThings, for instance, any developer may write an app in SmartThings SDK (based on a programming language named Groovy (Groovy)), 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 sensors and hubs).
The openness leads to the rapid adoptions of such IoT platforms, however, it also brings some issues. An important issue 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 (SmartApp) shows that such inter-app conflicts are common: Among 22 randomly selected public SmartApps, we found 18 conflicting groups of apps, as detailed in Section 7. According to statistics, the number of smart connected homes could hit up to 700 million homes by 2020, rising from somewhere between 100 million and 200 million homes now, fueled by mass consumer adoption and an increase in the number of devices and apps available (Gartner, Inc.). As the most popular IoT application, close to 1.5 million IoT developers are currently working on Smart Home projects (VisionMobile). 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 in the systems are defined in the design stage of the systems by a team in a coordinated manner, the problem has been addressed through a specification-based method (Bieliık et al., 2015). The development of these systems often starts with some formal specifications of the design (e.g., through Esterel-like synchronous languages (Berry, 2000)), 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 (Matya et al., 2014). The specification-based method apparently cannot apply to open IoT platforms as arbitrary apps could be installed into them by the user.

How to address inter-app conflicts on open IoT platforms still remains preliminarily understood. Recent years have seen several efforts, but they have all considered some basic types of conflicts: two apps access the same device (e.g., in HomeOS (Dixon et al., 2012)) or have different/opposite effects on the same device (e.g., in SIFT (Liang et al., 2015)) or environment (e.g., DepSys (Munir & Stankovic, 2014)).

In this work, we argue that the previous definitions of inter-app conflicts are imprecise, and subsequently, the conflict detection methods that they have developed upon the definitions are inadequate. We further provide an improved definition and categorization of IoT inter-app conflicts. We classify inter-app conflicts into strong conflicts and weak conflicts, depending on whether certain functionalities or control of an app get disabled or affected by another app. Moreover, we uncover the category of implicit conflicts among apps that work on different devices. These understandings prepare the foundation for the analysis and detection of inter-app conflicts. (Section 3)

The rectified definition offers a more comprehensive coverage of conflicts. As a result, the detections of the conflicts demand more sophisticated treatments than prior studies have given. The second part of this paper presents our solution. A key component of the solution is inter-app ER Nets (IA Nets), a novel representation of IoT apps and their interplays. IA Nets is based on Petri Nets (Ghezzi et al., 1991), a classical formalism for real-time system specifications. We equip IA Nets with a set of novel features such that it can concisely represent the controls in each IoT app and the various schedules of events, and at the same time, capture all the interplays among apps, allowing easy inferences of the various inter-app conflicts. (Section 4.1)

The third major contribution of this paper is an automatic inference algorithm for detecting inter-app conflicts, built upon the IA Nets representation of apps. The inference is based on first-order logic, and centers around two fundamental theorems on inter-app conflicts and a set of device models. It efficiently detects both strong and weak conflicts that are explicit or implicit. (Section 4.2)

The proposed techniques are general, applicable to various IoT platforms. For evaluation, we focus on the SmartThings platform. As a scripting language-based platform, SmartThings allows more flexible programings of IoT controls than rule-based platforms (e.g., IFTTT (IFTTT)) do, and hence give more challenges for conflict detections. We develop a tool named DIAMond (for Detection Inter-App conflicts), which integrates all our proposed techniques together, along with a compiler for automatically constructing IA Nets from SmartThings apps. We apply DIAMond to 22 public SmartThings apps. The experiments demonstrate that DIAMond successfully detects all types of conflicts among those apps with marginal time overhead. ...
2. Premises and Background

This section describes some premises and background that are necessary for understanding the rest of the paper.

Focus The problem focused in this work is inter-app control conflicts. Debugging issues within an app (e.g., those caused by bugs in implementing a certain functionality) has been studied in some prior work (Croft et al., 2015; Liang et al., 2016). They are outside the scope of this work. This work takes apps as they are, trying to find control conflicts among them.

IoT Programming There is a variety of ways to program an IoT app, depending on the used IoT platforms. They in general fall into two categories: rule-based or scripting language-based. The former is represented by If-This-Then-That (IFTTT) (IFTTT), in which, app developers just need to write a few high-level rules. The latter is represented by Samsung SmartThings (SmartThings), in which, app developers write apps in an SDK that is built upon scripting language Groovy (Groovy). The rule-based methods are easy to use by general users, while the scripting languages are Turing complete, allowing more flexibility in programming, which is essential for complicated 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. We next provide some background on SmartThings.

SmartThings SmartThings is a representative of open IoT platforms, with an architecture similar to other open IoT platforms. At a high level, SmartThings offers a distributed platform as depicted in Figure 1. It provides an SDK for common programmers to develop apps that can run upon SmartThings environments, controlling compatible devices. Such apps are called SmartApps.

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 a software package based on a scripting language Groovy, equipped with some libraries special to SmartThings. Figure 2 shows the code of a simple SmartApp. As most SmartApp, the sample app contains four key components.

(1) The first part uses a “definition” segment to express some meta data of the app, such as its name, author, description and so on. It determines how the app is described in the user interface (e.g., on the smartphone) after it is installed.

(2) The second part uses a “preferences” segment to define what devices and other options are required to install the app. 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 will first list all the devices with the “contact-Sensor” 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 physical devices to be controlled with the variables (“contact1” and “light1" in our example) in the app.

(3) 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 that status of device “contact1” turns to “contact.open” in SmartThings, when a device with a “contact-Sensor” capability opens, its status in the SmartThings runtime turns to “contact.open” automatically). 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. The “updated” method is called when the preferences of an installed app is updated. The “unschedule" method removes all scheduled executions for this app.
A Systematic Examination of Inter-App Conflicts Detections in Open IoT Systems

//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.
definition(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(){}

def(){}

def(){}

//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()

Figure 2. A SmartApp.

(4) 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 “on-Handler” method, the cloud 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. The call of “runIn” in the example invokes method “offHandler” 5min after the call of the “runIn” method.

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 DIAmond platform.

As most other IoT systems, the SmartThings platform utilizes an asynchronous execution model. Every device control method call is asynchronous: the execution of the program continues without waiting for the call to return. The design is to help achieve an overall good responsiveness as some device control may take a while to finish due to the network or device delays.

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 according to the app. All events are asynchronous.

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 detections of inter-app conflicts.

(1) Definition 1: Two apps conflict if they access the same device at the same time (used in HomeOS (Dixon et al., 2012)).

(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 (Liang et al., 2015).

(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. The definition, to a certain degree, resembles what DepSys (Munir & Stankovic, 2014) uses.

The three definitions form an evolving path, getting increasingly more precise. 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

---

1DepSys 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 apps directly.
step further; it considers some conflicts by two different devices. An example is that one app turns on a dehydrator while the other turns 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

This part introduces our new definition of inter-app conflicts. The new definition addresses the issues of the previous definitions, and at the same time, classifies conflicts into different categories, which could help the creation of discriminative treatments to the conflicts of different seriousness.

Specifically, we define inter-app conflicts in two main categories: strong conflicts and weak conflicts.

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

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 two apps run together, there is a weak conflict if the apps control certain actuator differently, while no app’s actions get disabled as a result of the interactions of the apps.

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.

In general, strong conflicts are problematic as they reduce the functionalities of some app. It is not always so clear for weak conflicts. For instance, consider two apps: one turns on a siren when window1 is opened at night, the other turns on the same siren when smoke is detected. Each of the apps affects the way the siren is controlled. However, the two controls may be both wanted by the user. Overall, whether a weak conflict is problematic is subject to the particular user. For that difference, our detection tool distinguishes these two kinds of conflicts; the system may then give different treatments to the two kinds of conflicts.

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. Therefore, when the two apps run together, some picture takings could get missed. To detect such implicit conflicts, it 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 survey and experiments, including all the aforementioned conflicts that previous definitions cannot cover.

4. Conflicts Detection

When the definition of conflicts is narrow, the detection can be simple. For instance, for the conflicts defined in Defini-
tion 1, the detector needs to check only the set of devices each app accesses as HomeOS does (Dixon et al., 2012). 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. To assist the reasoning, we design a novel 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: IA Nets

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 be flexible enough to express the operations of various devices, the conditions for these operations to get triggered, and the consequences of the operations. The formalism should capture all the essential interplays of different apps in a single IoT system. It at the same time should be amenable for automatic inferences of all possible conflicts between apps.

The formalism we design is named *inter-app ER Nets* (IA Nets), which is an extended form of Petri Nets. Petri Nets is a form originally proposed for representing chemical processes, and has been later used broadly in specifying real-time systems (Reisig, 1985; Peterson, 1981). Petri Nets has many variations. Our formalism is an extension to the basic Environment/Relationship (ER) Nets (Ghezzi et al., 1991).

ER Nets are high-level Petri Nets. As illustrated in Figure 3 (a), ER nets consist of circles and boxes and edges among them. The circles are called *places*, with each containing some variables; the box is called a *transition*, corresponding to some action. The incoming edges of a transition indicate the places relevant to the triggering of the action (those places are called the *preset* of the transition), while the outgoing edges of a transition indicate which places get affected by the action (those places are called the *postset* of the transition). Each transition is associated with a function indicating what values the variables in the preset should have for the transition to fire, and what values the postset variables will get after the transition. A formal definition of ER Nets can be seen in some earlier papers (Ghezzi et al., 1991). ER Nets have been used to specify closed real-time systems.

Our design of IA Nets adds several features to the basic ER Nets to help express and analyze inter-app relations. We describe these features by drawing on the example in Figure 3 (b).

(1) There are three kinds of places, respectively denoting the capabilities of devices, the statuses of scheduled functions, and the global variables in an app respectively. The concatenation of the device ID and the capability forms the place ID of each place corresponding to a capability of a certain device installed in the target IoT environment. Figure 3 (b) lists the IDs of all the places: the “contact” capability of two contact sensors, the “sound” capability of a siren and a music player. Two apps operate on the same capability of the same device would share the same place in their parts of the Nets. The place representing the scheduled function use function ID as its place ID. This kind of place has two attributes: flag and schetime. “flag” has two values, true and false, denoting the function is scheduled or unscheduled. “flag” changes its value once “schedule” or “unschedule” is called. “schetime” records the schedule time of the function which is the first variable in “schedule” API. Global variables in an app are used to remember information across executions. The concatenation of the app ID and the variable name forms the ID of a place corresponding to a global variable in an app. As ER Nets, a place in IA Nets may have other attributes, such as “chronos” for representing time (Ghezzi et al., 1991). Global variables in an app are used to remember information across executions.

(2) A special attribute is added to each place to represent its influence zone. Influence zone is a new concept we introduce to help deal with some implicit interplays among apps. It refers to the attribute (e.g., temperature, brightness, sound) of an area (e.g., a room) that is physically

![Figure 3. Examples of ER Nets and IA Nets.](image-url)
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. Some capabilities of some devices (e.g., picture taking by a camera, and contact sensors in Figure 3 b) do not create a physical influence to the environment. Their influence zone is just the capabilities themselves and is not explicitly listed. Influence zone is needed because even though two apps do not operate on the same device, they could still conflict if their influence zones overlap (as shown by the example of two lights in the same room as mentioned in Section 3). In IA Nets, we use a special kind of edges (influence edge) to connect two places that have overlapped influence zones. The dotted line between $P_1$ and $P_3$ in Figure 3 (b) illustrates the influence edge between the two places as their influence zones are both the sound of the same room. Section 4.2 will further elaborate this concept when describing device models.

(3) Each transition now carries an app ID (e.g., the superscript of $t$ in Figure 3 b) such that when multiple apps in an IoT environment are put together into one IA Nets, the transitions from different apps can be easily told apart.

(4) In IA Nets, a transition is composed of 4 sets: Preset places, Postset places, PreConditions set and PostConditions set. The function associated with PreConditions and PostConditions of a transition is written as a set of Horn clauses, with each in the implication form as “conditions $\Rightarrow$ 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. PreConditions is the set of transition firing conditions corresponding to values of Preset places. PostConditions is the set of firing consequences corresponding to Postset places. In our example in Figure 3 (b), the transition indicates that when either contact1.open or contact2.open is true, the siren should sound. If a place $p$ belongs to both Preset and Postset of a transition, $p$ and $p'$ denote the same place in Preset and Postset respectively.

IA Nets offers a way to represent the controls of multiple apps in a single form. The connections in IA Nets 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.

4.2. Inference for Detecting Conflicts

In this section, we describe the algorithm for detecting inter-app conflicts. We first introduce several terms and two fundamental theorems.

The detection works on the IA Nets representations of the apps. Recall that each transition in an IA Nets 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, formula

$$\text{switch} = 1 \implies \text{light} = 1$$

turns into

$$\text{light} \lor \neg \text{switch},$$

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: $\{\text{light}=1, \text{switch}=\ast\}$ and $\{\text{switch}=0, \text{light}=\ast\}$, where $\ast$ represents all possible values. For a subgraph of an IA Nets, the solution set to the conjunction of all the transitions in that subgraph is called the solution set of that subgraph.

### Fundamental Theorems

Our conflict detection is based on the following two fundamental theorems.

**Theorem 4.1 Strong Conflict Theorem.** For a set of apps $T$, let $F$ be the conjunction of all the functions contained in $T$, and $S$ be the solution set to $F$. $T$ contains a strong conflict if and only if there is at least one solution (denoted as $s$) to some app in $T$ that cannot be implied by $S$, that is, $S \not\models s$.

For instance, consider two apps with respective transitions as follows: $t_1$: $\text{switch}=1 \implies \text{light}=1$ in app1, $t_2$: $\text{switch}=1 \implies \text{light}=0$ in app2. Function $F$ would be $\text{light} \lor \neg \text{switch} \land (\neg \text{light} \lor \neg \text{switch})$. First-order logic simplifies it into $\neg \text{switch}$. The solution set $S$ is hence $\{\text{switch}=0, \text{light}=\ast\}$. It does not imply one of the solutions to app1: $\{\text{switch}=1, \text{light}=1\}$. Therefore, according to the Strong Conflict Theorem, the two apps have a strong conflict.

The correctness of the theorem comes directly from the definition of strong conflicts in Section 3. The intuition is that after putting the apps together, the set of behaviors allowed for one of the apps becomes smaller than before. Compared to the definition, the operational nature of this theorem offers a way for the development of conflict detectors.

Weak conflicts have a condition stricter than strong conflicts have. Before introducing the theorem on weak conflicts, we introduce a term control subgraph of a place. For a place $p$ in an IA Nets, its control subgraph is the subgraph
of the IA Nets, in which, every place 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 \( cg(P_3) \) contains only the top part consisting of \( P_2, P_3 \), and \( t_2 \). The control graph of a place determines how the capability in that place is controlled.

**Theorem 4.2** Weak Conflict Theorem. For a set of apps \( T \), there is a weak conflict if there is a place \( p \) 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 neither of the control graphs is empty.

The final condition in the definition ensures that both apps involve place \( p \) in their controls. We illustrate the theorem through the two example apps shown in Figure 4. Place \( P_3 \) appears in both IA Nets. Its control graphs in the two apps are respectively the entire IA Nets in Figure 4 (a), and the “\( P_2, t_2, P_3 \)” part of the IA Nets in Figure 4 (b). The solution sets of the two control graphs are respectively \{siren, -contact1\} and \{siren, -contact2\}. These two differ, indicating the differences in the control of \( P_3 \) and hence a weak conflict. The IA Nets representation provides conveniences for identifying the relevant control subgraphs and omitting the irrelevant controls for the conflict inferences.

**Device Modeling**

The two theorems are about conflicts on the same device. As we mentioned before, 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 1 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.

**Detection Algorithm**

We design an algorithm that uses the two theorems along with the device models to detect both strong and weak conflicts. The algorithm has two versions. The first is for batch detection, trying to detect the conflicts among a set of apps. The second is for installation-time detection, trying to detect conflicts when an app is being installed into a system that already holds some other apps. The latter can be considered as an incremental version of the former; it fits most common usage scenarios in an IoT system.

Figure 5 outlines the batch detection algorithm in pseudo-code.

The algorithm first finds all the places that either appear in more than one app or have influence connections with other app. We call them common places. The algorithm records in InfSets all the sets of places that have overlapped influence zones. It then gets the logic formula in each of the control graphs of a common place. For the transitivity of equivalence, it checks every formula against the first formula. If their controls of the place are not equivalent to each other, a weak conflict is reported on that place as per the Weak Conflict Theorem. A record is added into the weak conflict set WC. It then checks whether the conjunction of all the formulas of a place is compatible with each of the formulas (procedure “compatible()” in Figure 5), and if not, it regards that as a strong conflict as per the Strong Conflict Theorem. It then checks each set in InfSets. If it contains places from multiple apps, it checks the type of the influence zone. If it is weak, it considers the

### Table 1. Models of Influence Zones of Some Devices

<table>
<thead>
<tr>
<th>Capability</th>
<th>Influence zone</th>
<th>Type of conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>light.switch</td>
<td>room</td>
<td>brightness</td>
</tr>
<tr>
<td>thermo.switch</td>
<td>room</td>
<td>temperature</td>
</tr>
<tr>
<td>siren.sound</td>
<td>room</td>
<td>sound</td>
</tr>
<tr>
<td>musicplayer.play</td>
<td>room</td>
<td>sound</td>
</tr>
<tr>
<td>camera.picTaking</td>
<td>lens</td>
<td>availability</td>
</tr>
<tr>
<td>camera.videoTaking</td>
<td>lens</td>
<td>availability</td>
</tr>
<tr>
<td>contact.open</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>thermo.switch</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 4. Two apps with weak conflicts on \( P_3 \).
Input:
N: a set of IA Nets of some apps;

Output:
SC: strong conflicts;
WC: weak conflicts;
SCI: strong conflicts due to influence zones;
WCI: weak conflicts due to influence zones;

Algorithm:

// Fill P and InfSets. P: set of places that each appear
// in multiple apps or have influence connections with
// other apps InfSets[indx[p]]: the set of places
// having an influence zone overlapping with p's
findAllSharedPlaces(N, &P, &InfSets);

foreach p in P
for i=1 to |N|
g=getControlGraph(p, N[i]);
p.f[i] = getFormula(g);
end for
// detecting weak conflicts on the same capability
for i=2 to |N|
if p.f[i] is enabled by f1
WC.add(p.i); // detecting strong conflicts on the same capability
end for

// checking if f1 is disabled by f2
if (∃p, q in s, p!=q & app(p)!=app(q) & r⇒p∧q)
SCI.add(s);
else if (s consists of places from multiple apps)
if (s.InfType == WEAK)
WCI.add(s);
else if (s.InfType == STRONG)
r = conjunction of f* of all the places in s
if (∃p, q in s, p=q & app(p)=app(q) & r⇒p∧q)
SCI.add(s);
end for

end for

end for
// detecting influence zone caused conflicts

Figure 5. Inter-App Conflict Detection Algorithm.

set forms some weak conflicts; if it is strong, it checks to see whether there are two different places from that set that belong 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(f1, f2)” used in the strong conflict check in Figure 5 is worth some more explanations. It checks whether there is a solution of f2 that cannot be implied from any solution of f1—that is, whether some function of f2 is disabled by f1. The check can be done through a SAT solver (e.g., Z3 (Z3)) by checking whether f2 ∧ (¬f1) can be satisfied. If so, it means that there is an solution that makes f2 true but f1 false. There hence has a strong conflict. (To avoid interferences from irrelevant variables, before the check, variables appearing in neither the postset nor the preset of f2 are removed from f1.)

The algorithm involves the proving of about $2 \sum_{i=1,P} n(i)$ formulas, where $|P|$ is the number of common places and $n(i)$ is the number of apps sharing the $i^{th}$ common place. 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 place 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. The algorithm is similar to the one shown in Figure 5 except that it only needs to check whether the newly added app conflicts with existing ones. It takes even less time to do.

Discussion about 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. IA Nets Construction

This section explains how IA Nets can be constructed from IoT apps. As a general formalism, IA Nets is applicable to IoT apps written in either high-level rules or low-level scripting languages. Building IA Nets is in general more complicated for apps written in scripting languages than in high-level rules, as the apps can be more complex for the more expressiveness of the scripting languages than rules. To help better cover the challenges, this section takes SmartThings apps (in Groovy scripting language) as the example to explain the process of IA Nets construction.

When an app is installed in the system, places are created for devices, scheduled functions and global variables; transitions are constructed based on data flow and control flow analysis. The generation of IA nets focuses on the controls
of devices and device triggering relations in the apps. It ignores complicated computations in the code. The results of the complicated computations could have two kinds of impact on device controls: (1) they are used in condition statements to determine which branch of a condition statement is taken; (2) they are used to set the numerical values of some devices (e.g., temperature for a thermometer). In the generated IA Nets, symbolic notions (i.e., variables) are used to represent these computation results in the transition functions.

**SmartApp API** Device control flow is mostly derived from SmartApp APIs (e.g., “subscribe”, “schedule”, “unschedule”, “runIn”). Some of the transition construction rules for these APIs are shown in Table 2. There are other scheduling methods that create recurring schedules, such as “runOnce”, “runEvery5Minutes”, “runEvery10Minutes”, “runEvery15Minutes”, “runEvery30Minutes”, “runEvery1Hour”, “runEvery3Hours”. 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”.

A special kind of devices is sensor. Unlike other devices, a sensor needs no trigger to act. Its state is automatically sensed in every $x$ seconds ($x$ is a parameter). Its transition function hence tracks the latest state and time.

### Condition Statements and Loops

Transition construction rules are also created for condition statements and loops. (1) **Condition statements.** For a condition statement, the capabilities of devices appearing in its branches are represented as places in IA Nets 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 places. Our conflict detector, presented in the next subsection, works conservatively by assuming that the branches are all possible to be taken. (2) **Loops.** Loops are rarely used in SmartApps. Repetitive actions are usually materialized by “schedule” kind of APIs. We observe only one use of loops in 22 public SmartApps, which lets a camera take photos repeatedly for 5 times with a 5min delay in the between. In general, symbolic variables are used in the postcondition of the corresponding transition function to express the impact of the loop iteration on the device control. In the camera example, the postcondition is written as

$$\{\text{camera}.picTaking & \text{camera}.chronos = \text{camera}.chronos + i * 5 * 60(i = 0, 1, 2, 3, 4)\}.$$

### 6. DIAmond Infrastructure

We develop a software framework named DIAmond that incorporates all the proposed techniques. It consists of some extra features to serve as an infrastructure to allow automatic analysis of multiple SmartThings apps and their in-

<table>
<thead>
<tr>
<th>API with Description</th>
<th>Preset</th>
<th>Postset</th>
<th>PreConditions</th>
<th>PostConditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td>the place denoting the capability of the triggering device erDev</td>
<td>places denoting the capability of the triggered devices in subFunc</td>
<td>“erDev.value”</td>
<td>change the status of the triggered devices as per subFunc</td>
</tr>
<tr>
<td>schedule(scheTime, scheFunc): Execute scheFunc every day at scheTime</td>
<td>places denoting the triggers of “schedule”</td>
<td>the place denoting scheFunc</td>
<td>the status of the triggering devices</td>
<td>“scheFunc.flag = true”</td>
</tr>
<tr>
<td>unschedule(scheFunc): unschedule the function scheFunc</td>
<td>places denoting the triggers of “unschedule”</td>
<td>place denoting scheFunc</td>
<td>the status of trigger devices</td>
<td>“scheFunc.flag = false”</td>
</tr>
<tr>
<td>runIn(runInTime, runInFunc): execute runInFunc after runInTime seconds from now</td>
<td>places denoting the triggers of “runIn”</td>
<td>places denoting the capabilities of the triggered devices in runInFunc</td>
<td>the status of trigger devices</td>
<td>Set the status of the triggered devices and set the “chronos” of triggered devices as scheFunc.chetime</td>
</tr>
</tbody>
</table>

#### Table 2. Some of SmartApp API Construction Rules
terplays. As the SmartThings platform is not open-source, it is not directly usable. DIAmond is based on an open-source Groovy Engine (Groovy).

Figure 6 shows the high-level structure of DIAmond. It consists of the following three major components, represented by the three 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. These SDK are not open-source. To circumvent the problem, we create a set of classes to emulate the functionalities of those support. These classes include capability classes, device-Type 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.

The preprocessor changes the input SmartApp to link them with these customized classes, making the app able to be complied and executed on the standard Groovy Engine. Such transformations make it possible to use open-source tools to analyze and execute SmartApps.

(2) Conflict Detector. This component takes multiple preprocessed SmartApps as input, derives the IA Nets 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. Both the model library and the detection algorithm are described in Section 4.2.

(3) Emulator. The third 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.

7. Evaluation

We collect 22 public SmartApps from SmartThings Github repository. There are 34 devices of 21 types, involving 22 capabilities. We use Z3 (Z3) as the SAT solver.

The left three columns in Table 3 show the description of each SmartApp and the involved devices and capabilities.

The fourth and fifth columns in Table 3 report the numbers of detected strong and weak conflicts. In the parentheses in those two columns of each row, 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. The rightmost column reports the device capabilities that are involved in the conflicts.

Overall, the tool detects 7 pairwise strong conflicts and 11 groups of weak conflicts in Table 3. Each weak conflict group contains 2 or 3 apps. Our manual examination of the source code verifies that these detected conflicts are all true conflicts of the right type. Even though our technique makes some conservative assumptions on unknown variables and branches and hence cannot guarantee a perfect precision. However, our experiments show that IoT apps are typically simple, plausibly determined by their nature of lightweight device controllers. Our tool hence produces no false alarms. The device capabilities involved in the conflicts are of a wide variety, ranging from carbon monoxide detector to locks, contact sensors, power meters, water sensors, and so on. We next use several case studies to provide some discussions on the conflicts.

Case 1: Conflicts on different devices with different influence zones.

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 on and closes the door if the thermostat is off.

DIAmond 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 the open of the window meanwhile 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. Other controls in these apps form another strong conflict, which is omitted from the discussion.

Case 2: Conflicts on different devices with the same influence zone.
App4 and App7 are both about light controls, but on two different lights in a single room. They illustrate a conflict involving different devices. App4 turns light1 on when contactSensor2 opens and App7 turns light4 on when vibration is detected by accelerationSensor2. Light1 and light4 happen to be in the same room and hence have the same influence zone. DIAmond detects such a weak conflict based on the influence edge between them in the IA Nets and the device model in Table 1.

Case 3: Conflicts on the same capability of the same device.

Figure 7 shows the core logic of App3, in which the set of randomly selected lights are turned on every frequencyTime on vacationDate. App18 turns lights off when no motion or presence is detected after a certain time. Once the randomly selected lights are turned on by App3, App18 will turn them off within delayMins. These two apps form some weak conflicts. DIAmond detects that the two apps share the places corresponding to the states of the lights. The control graphs of the places in the two apps have different solutions, indicating weak conflicts. For App21 and App22 in Case 1, there is also a weak conflict (different controls on the door), which is detected by DIAmond in a similar manner.

Case 4: Conflicts on different capabilities of the same device.

In this case study, we discuss two conflicts on different capabilities of the same device, with one as a strong conflict, the other as a weak conflict.

The first is the conflicts 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. The two capabilities of the camera2, picture capturing and video taking, conflict. When camera2 is busy recording video, no photo can be taken by this camera during the video time. If motionSensor1 is active after contactSensor2 opens and the video recording lasts 1 minute, the image of the active motion cannot be captured. DIAmond detects such strong conflicts through the device models. The combined IA Nets of the two apps have an influence edge connecting the two capabilities. Through the device models shown in Table 1, DIAmond immediately detects the strong conflicts.

Device bulb has two capabilities: ColorControl and ColorTemperature. Figure 8 shows App19, in which the bulb can be set to different colors when contact opens or mo-
The adapted DepSys method detects 4 out of 7 strong conflicts and 56% weak conflicts. Many of them involve controls of different devices that have same effects on the environment. For instance, the alarm in App6 and the music player in App9 both create sound in a room. They form a weak conflict for the influence they create, but the adapted DepSys misses it when they control the same device. An example is App5 and App18 in a setting that both play music for a period of time. The adapted 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.

DIAmond significantly outperforms previous methods for its more rigorous definitions of conflicts and its effective detection methods. It uncover all conflicts in those apps without giving any false alarms.

Comparison with Previous Methods We compare DIAmond's detection recall and precision to those of adapted DepSys (Munir & Stankovic, 2014), SIFT (Liang et al., 2015) and HomeOS (Dixon et al., 2012) on the 22 apps. Table 4 reports the results. As mentioned earlier, 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 Smart-Things 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 certain 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 4 out of 7 strong conflicts, 14 out of 34 true weak conflicts and gives 3 false conflicts. Two of the three missed strong conflicts are similar to the seeHouse 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 56% weak conflicts. Many of them involve controls of different devices that have same effects on the environment. For instance, the alarm in App6 and the music player in App9 both create sound in a room. They form a weak conflict for the influence they create, but the adapted DepSys does not recognize it as their effects are the same to the environment. The three false conflicts 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 in a period of time of a certain length. The adapted 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.

DIAmond significantly outperforms previous methods for its more rigorous definitions of conflicts and its effective detection methods. It uncovers all conflicts in those apps without giving any false alarms.

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 on the Intel machine and less than 0.2s on the Smartphone.

Most time in the conflict detection is spent on the execution of the detection algorithm; the construction of IA Nets and other operations take negligible time. IoT apps are typically small. In all the apps we collected, the code ranges from 60 to 400 lines of source code, with many lines being meta info (e.g., description of the app). Because the IA Nets only capture code related with device capabilities, the graphs 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 places and $n(i)$ is the number of apps sharing the $i$th common place. In our collection of the 22 apps, $|P|$ equals 16, and $n(i)$ is no more than 7 (contact-Sensor.open is the most popular place). The longest formula involves 10 device capabilities while most contains much less than that. So the detection finishes quickly.

How to best resolve conflicts is out of the scope of this paper. Here, we give a brief discussion. After detecting conflicts, there may be many possible ways to get them addressed. For instance, the tool may show some warn-
### Table 3. Inter-app Conflict Detection Results

<table>
<thead>
<tr>
<th>SmartApp No</th>
<th>Description of SmartApp</th>
<th>Device Involved</th>
<th>Number of Conflicts</th>
<th>Capability Involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Turn on a switch when CO2 levels are too high.</td>
<td>Co2Sensor, switch1</td>
<td>Strong: 1(6)</td>
<td>CarbonMonoxideDetector, Switch</td>
</tr>
<tr>
<td>2</td>
<td>Make sure a door is locked at a specific time. Option to add door contact sensor to lock if closed.</td>
<td>contactSensor1, lock</td>
<td>Weak: 2(11,15)</td>
<td>ContactSensor, Lock</td>
</tr>
<tr>
<td>3</td>
<td>Randomly turn on/off lights to simulate the appearance of an occupied home while you are away.</td>
<td>presenceSensor1, presenceSensor2, presenceSensor3, light1, light2, light3</td>
<td></td>
<td>PresenceSensor, Light</td>
</tr>
<tr>
<td>4</td>
<td>Turn on the half light if someone comes home (presence) and the door opens.</td>
<td>presenceSensor1, contactSensor1, light1, light4</td>
<td>Strong: 1(3)</td>
<td>PresenceSensor, ContactSensor, Switch</td>
</tr>
<tr>
<td>5</td>
<td>Turn your lights off after a period of no motion being observed.</td>
<td>motionSensor1, light1, light2, light3</td>
<td>Weak: 2(11,13)</td>
<td>MotionSensor, Switch</td>
</tr>
<tr>
<td>6</td>
<td>Turn on Monitor and siren the alarm when vibration is sensed.</td>
<td>accelerationSensor1, switch1, alarm</td>
<td></td>
<td>AccelerationSensor, Switch, Alarm</td>
</tr>
<tr>
<td>7</td>
<td>Turns on a light when a sensor is vibrated.</td>
<td>accelerationSensor2, light4</td>
<td></td>
<td>AccelerationSensor, Switch</td>
</tr>
<tr>
<td>8</td>
<td>Turns off device when wattage drops below a set level after a set time.</td>
<td>powerMeter, thermostat1</td>
<td></td>
<td>PowerMeter, Thermostat</td>
</tr>
<tr>
<td>9</td>
<td>Set up a reminder so that if you forget to take your medicine (determined by whether a cabinet or drawer has been opened) by specified time and music player plays a voice reminder.</td>
<td>contactSensor2, musicPlayer</td>
<td></td>
<td>ContactSensor, MusicPlayer</td>
</tr>
<tr>
<td>10</td>
<td>If the heating or cooling system come on, it gives notice if there are any windows or doors left open.</td>
<td>window1, window2, thermostat2, door</td>
<td></td>
<td>WindowShade, Thermostat, DoorControl</td>
</tr>
<tr>
<td>11</td>
<td>Turns on selected device(s) at a set time on selected days of the week only if a selected person is present and turns off selected device(s) after a set time or receive a lighting notification.</td>
<td>presenceSensor1, thermostat1, lock, garageDoor, light2, switch2, bulb</td>
<td>Strong: 2(15)</td>
<td>PresenceSensor, Thermostat, Lock, GarageDoor, Switch, ColorTemperature, ColorControl</td>
</tr>
<tr>
<td>12</td>
<td>Take a burst of photos when contact opens, vibration detected, motion or presence sensed or switch turns on.</td>
<td>contactSensor1, accelerationSensor1, motionSensor1, presenceSensor1, switch2, camera1, camera2</td>
<td>Weak: 2(11,11)</td>
<td>AccelerationSensor, ContactSensor, MotionSensor, Switch, ImageCapture</td>
</tr>
<tr>
<td>13</td>
<td>Select locks, presence, motion, acceleration or contact sensors to control a set of lights.</td>
<td>motionSensor1, motionSensor2, motionSensor3, presenceSensor1, presenceSensor2, accelerationSensor1, contactSensor1, contactSensor2, lock, light1, light2, light3, light4</td>
<td></td>
<td>MotionSensor, AccelerationSensor, ContactSensor, Switch</td>
</tr>
<tr>
<td>14</td>
<td>Close a selected valve if moisture is detected.</td>
<td>waterSensor, valve</td>
<td></td>
<td>WaterSensor, Valve</td>
</tr>
<tr>
<td>15</td>
<td>When door opens, change the bulb color and take a video for safeguard.</td>
<td>contactSensor2, bulb, camera2</td>
<td></td>
<td>ContactSensor, ColorControl, ColorTemperature, VideoTaking</td>
</tr>
<tr>
<td>16</td>
<td>Turn on the switch when someone has presence.</td>
<td>presenceSensor2, switch2</td>
<td></td>
<td>PresenceSensor, Switch</td>
</tr>
<tr>
<td>17</td>
<td>Turn on the warning switch when smoke is detected.</td>
<td>smokeDetector, switch2</td>
<td></td>
<td>SmokeDetector, Switch</td>
</tr>
<tr>
<td>18</td>
<td>Turn lights off when no motion and presence is detected for a set period of time.</td>
<td>motionSensor1, motionSensor2, presenceSensor1, presenceSensor2, light1, light4</td>
<td></td>
<td>MotionSensor, PresenceSensor, Switch</td>
</tr>
<tr>
<td>19</td>
<td>Sets the colors and brightness level of your Philips Hue lights to match your mood when contact opens or motion is active.</td>
<td>contactSensor2, motionSensor2, bulb</td>
<td></td>
<td>ContactSensor, MotionSensor, ColorControl, ColorTemperature</td>
</tr>
<tr>
<td>20</td>
<td>Sets your bulb to a specific color temperature with default Moonlight (4100) when a button is pushed.</td>
<td>button, bulb</td>
<td></td>
<td>Button, ColorControl, ColorTemperature</td>
</tr>
<tr>
<td>21</td>
<td>Close the door if any window opens.</td>
<td>door, window1, window2</td>
<td></td>
<td>DoorControl, WindowShade</td>
</tr>
<tr>
<td>22</td>
<td>Open the door if thermostat is off and close the door if thermostat is on.</td>
<td>door, thermostat1</td>
<td></td>
<td>DoorControl, Thermostat</td>
</tr>
</tbody>
</table>

**A Systematic Examination of Inter-App Conflicts Detections in Open IoT Systems**
ings to the user and ask them to decide whether to disable some apps. A more sophisticated method is to automatically modify the apps to fix the conflicts based on some feedback from the user. The problem is worth future studies.

8. Related Work

Recent years have seen increasing interest in home automation and other IoT applications. The work closely related with this study includes HomeOS (Dixon et al., 2012), SIFT (Liang et al., 2015), and DepSys (Munir & Stankovic, 2014). HomeOS (Dixon et al., 2012) 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. As Section 3.1 mentions, this definition is imprecise and subject to some important limitations. SIFT (Liang et al., 2015) proposes a rule-based programming platform for IoT. It gives some valuable discussions on the safety of IoT apps including inter-app conflicts. However, as Section 3.1 discusses, due to the preliminary definitions of inter-app conflicts that SIFT uses (Definition 2), its conflicts detection method has a limited coverage. DepSys (Munir & Stankovic, 2014) 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. Its definition of conflicts is similar to Definition 3 mentioned in Section 3.1, which 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.

Compared to these work, this paper provides a more rigorous definition of inter-app conflicts. It proposes IA Nets, the first high-level formalism to represent IoT controls and interactions, and presents a new algorithm for detecting the more comprehensive set of inter-app conflicts.

Table 4. Recall and precision of conflicts detections

<table>
<thead>
<tr>
<th>Method and Authors</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAmond</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Adapted DepSys (Munir &amp; Stankovic, 2014)</td>
<td>44%</td>
<td>85%</td>
</tr>
<tr>
<td>SIFT (Liang et al., 2015) (Definition 2)</td>
<td>29%</td>
<td>100%</td>
</tr>
<tr>
<td>HomeOS (Dixon et al., 2012) (Definition 1)</td>
<td>63%</td>
<td>84%</td>
</tr>
</tbody>
</table>

Conflicts could happen in traditional real-time/reactive systems (Davis & Burns, 2011; Khurshid et al., 2013). 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 (Berry, 2000)), 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.

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 (Aviram et al., 2012; Kasi & Sarma, 2013; Boutin et al., 2014). The problems tackled in this work are about detecting whether one app’s control on a device capability gets affected by another app.

There is a body of work on testing the implementation of a real-time system to determine whether it conforms with the design of the system in terms of timing constraints. The studies either use static code analysis to estimate the worst-case and best-case execution times of a task code (Puschner & Koza, 1989; Braberman et al., 1997), or use dynamic testing (e.g., evolutionary testing (Wegener & Grochtmann, 1998; Wegener & Mueller, 2001) that searches for test data producing extreme execution times) to actually examine the temporal behavior of the real-time systems. Our anal-
ysis considers some temporal aspects of the apps that are closely related with inter-app conflicts; it is not to find out whether the execution times of the apps conform certain designs—which are not available for such open IoT systems.

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 (Hsiao et al., 2014). They concentrate on data races, rather than inter-app conflicts.

Petri Nets was originally proposed by Carl Adam Petri (at age of 13) for the purpose of describing chemical processes. It has been used for specifying and analyzing the designs of real-time systems (Reisig, 1985; Peterson, 1981). Various Petri Nets formalisms have been developed (Berthomieu & Menasche, 1983; Ramachandran, 1974; Richter, 1986; Ramamurthy & Ho, 1980; Merlin & Farber, 1976; Leveson & Stolzy, 1987) for correctness and performance analysis of real-time systems. The high level Petri Nets used in this work was based on a form proposed by Ghezzi and others for describing time-critical systems (Ghezzi et al., 1991).

9. Conclusions

This paper has presented a systematic study on detecting various conflicts among IoT apps in an open IoT platform. It gives a more precise characterizations of inter-app conflicts than previous studies do. It provides formal definitions of inter-app strong conflicts and weak conflicts, and introduces the concept of implicit conflicts caused by influence zone of IoT devices. It proposes a formalism named IA Nets to represent the controls of devices by IoT apps. It develops some fundamental theorems and algorithms for detecting inter-app conflicts upon IA Nets. It incorporates all these techniques into an open-source infrastructure named DIAmond for analyzing SmartThings apps. Experiments show that the techniques are effective in detecting various inter-app conflicts. On 22 public SmartThings apps, it successfully detect 7 pairwise strong conflicts and 11 groups of weak conflicts with no false alarms.

IoT is developing fast. CISCO estimates an average of 6.6 devices per person leading to 50 billion devices in 2020 (Cisco). A recent report (VisionMobile) states that IoT developers have reached 4.5 million in 2015, and 1.5 million of them are focused on smart home apps. As the trend continues, we expect that inter-app conflicts will become increasingly common. The techniques developed in this work show the promise for meeting the needs. We plan to release DIAmond as an open-source infrastructure for the community to use for further explorations of this important problem.

Acknowledgments

This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grant No. 1547105, 1455404, 1455733 (CAREER), and 1525609. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE and NSF.

References


Dixon, C., Mahajan, R., Agarwal, S., Brush, A.J., Lee, B., Saroiu, S., and Bahl, P. An operating system for the
A Systematic Examination of Inter-App Conflicts Detections in Open IoT Systems


