DepSys: Dependency Aware Integration of Cyber-Physical Systems for Smart Homes

Sirajum Munir
Department of Computer Science,
University of Virginia
Charlottesville, VA, USA
munir@cs.virginia.edu

John A. Stankovic
Department of Computer Science,
University of Virginia
Charlottesville, VA, USA
stankovic@cs.virginia.edu

ABSTRACT

As sensor and actuator networks mature, they become a core utility of smart homes like electricity and water and enable the running of many CPS applications. Like other Cyber-Physical Systems (CPSs), when a number of applications share physical world entities, it raises many systems of systems interdependency problems. Such problems arise in the cyber part mainly because each application has assumptions on the physical world entities without knowing how other applications work. In this work, we propose DepSys, a utility sensing and actuation infrastructure for smart homes that provides comprehensive strategies to specify, detect, and resolve conflicts in a home setting. Based on real home data, we demonstrate the severity of conflicts when multiple CPSs are integrated and the significant ability of detecting and resolving such conflicts using DepSys.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems

General Terms

Algorithms, Measurement, Experimentation

Keywords

Dependency, Control, Conflict Detection, Conflict Resolution

1. INTRODUCTION

One vision for Cyber-Physical Systems (CPSs) in home environments is that an underlying sensor and actuator network will act as a utility similar to electricity and water. Then, different CPS applications in domains such as health, security, entertainment, and energy can be installed on this utility. While each CPS application must solve its own problems, the sharing of a sensor and actuator utility across multiple simultaneously running applications can result in many systems-of-systems interference problems, especially with the actuators. Interferences arise from many issues, but primarily when the cyber depends on assumptions about the environment, the hardware platform, requirements, naming, control and various device semantics. Previous work, in general, has considered relatively simple dependencies related to numbers and types of parameters, versions of underlying operating systems, and availability of correct underlying hardware. This paper presents a design of a utility sensing and actuation infrastructure for a smart home that can run many CPS apps in many domains. It evaluates the design by using emulated apps and real in-home data. The full implementation of DepSys is future work.

It has been hypothesized that integrating systems in a smart home will have innumerable advantages. For example, let's assume that we integrate the systems responsible for energy management and home health care. Such integration will allow the energy management system to adjust room temperature depending on the physiological status of the residents as detected by the home health care system. Also, integration will avoid negative consequences. For example, the integrated system will not turn off medical appliances to save energy while they are being used as suggested by the home health care system. In addition to these advantages, all the systems can share sensors and actuators, which will reduce cost of deployment, improve aesthetics of the rooms, and reduce channel contention.

However, integrating multiple systems is very challenging as each individual system has its own assumptions and strategies to control physical world variables without much knowledge of the other systems, which

leads to conflicts when these systems are integrated without careful consideration.

In this work, we propose a design for a framework named DepSys that integrates various systems in a home by considering a comprehensive spectrum of dependencies and treating each system as an app. The integrated system, DepSys, is a Cyber-Physical System by its own nature, as sensing, communication, computation, and control are all present in the DepSys design. Individual systems use sensors for detecting environmental parameters and behaviors of the residents. Each system communicates with its sensors and actuators and performs computations for taking appropriate control decisions to actuate on the physical world entities, e.g., lights, HVAC, fire alarm, doors, windows, and computers.

This work has three major research contributions. First, to the best of our knowledge, DepSys provides the most comprehensive strategies to specify, detect, and resolve conflicts in a home setting by addressing a spectrum of dependencies including requirements, name, and control dependencies. In addition, DepSys handles the case when app developers fail to specify dependencies. Second, DepSys automatically resolves control conflicts of sensors and offers a strategy for resolving control conflicts of actuators that reduces the cognitive burden of the users and allows apps to be run in a more flexible way than the state of the art solution. Also, DepSys can detect conflict across devices by considering their impact on the environment, e.g., one app is running a humidifier and another app is running a dehumidifier at the same time, which state of the art solutions can't detect. Third, by using 34 days of data from a real home and using 35 apps from various categories, including energy, health, security, and entertainment, we demonstrate the severity of conflicts when multiple CPSs are integrated in a home setting and the significant ability of detecting and resolving such conflicts using the DepSys concepts.

2. RELATED WORK

Integrating systems in a home depends on the architecture of system integration. There have been a few architectures proposed for homes, including AlarmNet[20], Empath [12], and HomeOS [13]. We use an app based architecture for the integrated system of the home, motivated by their use in smart phones and HomeOS [13]. Although the DepSys architecture is similar to HomeOS, it has additional features for comprehensive dependency detection and resolution across apps.

Many commercial home automation and security systems [5] [7] [3] [2] are available on the market that integrate multiple devices in the home. However, these systems are usually monolithic, come with a fixed set of apps designed not to conflict with each other, and are not extensible for external app developers. Those that are extensible do not have sophisticated conflict resolution mechanism as found in DepSys. For example, HomeOS [13] provides a PC-like abstraction for technologies in the home and gives users a management interface designed for home environments. HomeOS uses app priority to resolve conflicts. Here is a simple example that shows that just priority is not enough for accurate actuation on appliances. Assume that a security app turns on light L1 at 8 PM and turns it off at 9 PM to discourage burglary. An energy app turns on lights when motion is detected and turns off lights when there is no motion for 10 minutes. If we set higher priority to the security app, then it is possible that it will turn off lights at 9 PM although there are people moving around. On the other hand, if we have higher priority to the energy app, then the energy app may turn off light L1 at 8:10 PM after detecting no motion. Our solution requires the app developers to specify additional metadata called *emphasis*, which allows DepSys to detect and resolve such conflicts more accurately.

There has been several works on running concurrent applications in sensor networks [15] [14] [9] [21] [17], but these works have limitations in resolving conflicting application requirements. For example, TinyCubus [15] offers a framework for running concurrent applications per network. However, a single sensor cannot be used by multiple applications at the same time. Melete [21] enables execution of concurrent applications on a single sensor node by hosting a virtual machine at node level. However, it ignores the problem of conflicting application requirements. PhysicalNet [18] [19] provides a generic paradigm for managing and programming worldwide distributed heterogeneous sensor and actuator resources. It uses resolvers to resolve conflicts. A resolver is a Java method that takes into account access rights, priority etc. to resolve conflicts. Owners can select resolvers from the library of PhysicalNet or they can implement their own resolvers. Comparing with these solutions, DepSys offers more fine grained dependency detection and resolution that these solutions cannot provide, e.g., conflict detection across appliances and differentiating false conflicts from true conflicts by considering device semantics.

3. SYSTEM ARCHITECTURE

Being motivated by the app based architecture in smart phones, we create a similar paradigm for smart homes. However, such a paradigm is more challenging for smart homes than for smart phones for several reasons. First, in a smart phone, the user usually interacts with one app at a time and that app usually gets the highest priority in resolving conflicts. However, in a home, multiple applications may be running simultaneously and it is not easy to decide how to resolve conflicts. Second,

smart homes have a lot more sensors and actuators than smart phones raising the severity of device level conflicts and device heterogeneity issues. Although an app based architecture for the home is not a completely novel concept as HomeOS [13] uses a similar architecture and addresses some device heterogeneity issues, but state of the art solutions lack capability in detecting and resolving conflicts that DepSys offers.

The DepSys system architecture is shown in Figure 1. App developers specify dependency information as meta data within their apps and their apps are put in an app store. Users can choose and install apps from the app store. DepSys uses the meta-data to detect and resolve conflicts at app installation time and at run-time. DepSys provides a platform for running the installed apps where apps are run at the top layer. Sensors (S1, S2, S3) and actuators (A1, A2) deployed in the home can be plugged into the system and the Device Plugins layer contains the device drivers[1]. We consider humans as sensors as they can specify their preferences and change app parameters. Before running an app, DepSys-Static Check (DepSys-SC) checks whether the deployment in the house satisfies the app requirement using two modules: HomeTree and Requirement Dependency Checker (RDP). HomeTree contains the positions of the deployed sensors and actuators of the house and RDP checks whether an app can be run for the current deployment of the house. The Preference Learning Module (PLM) learns user preferences and allows users to choose a policy. DepSys-Runtime Check (DepSys-RC) contains three modules: Actuator Control Dependency Resolver (ACDR), Sensor Control Dependency Resolver (SCDR), and Missing Dependency Checker (MDC). ACDR and SCDR resolve control dependencies of the actuators and the sensors, respectively, using user preferences and policies specified in the PLM. The MDC addresses the missing dependency problem, i.e., when the app developer does not specify a dependency that actually exists.

4. SPECTRUM OF DEPENDENCIES

In order to develop an app based paradigm for smart homes, we need to address a spectrum of dependencies. The dependencies we are listing in this section are not unheard of, but these dependencies must be addressed for the app based paradigm and the mechanisms to address some of the dependencies are novel. We clearly articulate the novelty when describing dependency addressing mechanism in the following sections. Note that our list of dependencies may not be complete. However, it is extensible. The dependencies are:

1. Requirement Dependency: It can be of 2 types:

[1]The role of the Device Plugins layer is similar to that of the Device connectivity and the Device functionality layers of HomeOS.

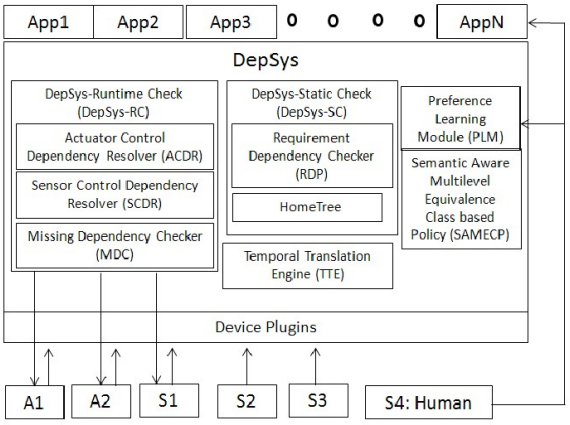

Figure 1: DepSys System architecture.

a. Requirements of the app: It addresses the requirement of an app to figure out if the app can be run in the house, e.g., *App1* may require that there has to be at least one motion sensor in every room or the acoustic sensor sampling rate has to be at least 4 KHz.

b. Requirements of the available resources (sensors/actuators): These are specified in DepSys and configured by the user/deployer. For example, the acoustic sensor *AC1* has to last for 15 days with the current available energy.

2. Name dependency: Name dependency needs to be addressed so that app requirement can be compared with a resource availability description to determine if an app can be installed and run in a particular deployment setting. Also, it is important for comparing across apps to determine whether the apps may conflict with each other or not.

3. Control dependency for the sensors: It arises when multiple apps want to control a sensor in different ways. For example, *App3* wants acoustic sensor *AC1*'s data at a rate that violates resource requirements in 1 (b), or *App1* wants accelerometer *ACC1*'s data at 50 Hz, whereas *App2* wants it at 100 Hz.

4. Control dependency for the actuators: It arises when multiple apps want to control an actuator or multiple actuators in a conflicting way. For example, *App1* detects depression and wants to turn on light *L1*, whereas *App2* wants to turn it off to save energy. Similarly, *App3* wants to run a humidifier and *App4* wants to run a dehumidifier at the same time.

5. Missing Dependency: It arises when app developers forget to specify the dependency information of their apps. For example, *App1* forgets to specify its dependency on light *L1* in its meta data, but at runtime it tries to control *L1*.

6. App interdependency: It arises when one app (dependent app) relies on another app (independent app). For example, *App1* announces residents' locations, i.e. localization information and some other apps listen to

the announcement and use this information to take action. If the independent app makes an error, the error may propagate and affect all the dependent apps and may cause a lot of conflicts in the system. Since DepSys doesn't know the internal logic of any app, it is extremely difficult for DepSys to detect such conflicts and that's why we do not address such dependency in this work. However, we address the other 5 dependencies.

4.1 Addressing Requirement Dependency

DepSys requires app developers to specify the requirement of each app in a manifest file written in XML. The reason for using XML is because it is simple, extensible, self-descriptive, and human-readable. Although describing app requirements in an XML file is not a novel idea, our novelty lies in determining the appropriate tags for a home setting and the design and use of a HomeTree (see later) for app compatibility checking. As an example, an energy management app that needs at least one motion sensor in all the rooms and at least one contact sensor on all the windows and doors describes its requirement as follows:

```
<requirement>
  <device>
    <device_type> X10_motion_sensor </device_type>
    <position> NULL </position>
    <container> room </container>
    <container_selection> all </container_selection>
    <device_count> at_least_1 </device_count>
    <level> strict </level>
  </device>
  <device>
    <device_type> X10_contact_sensor </device_type>
    <position> NULL </position>
    <container> window, door </container>
    <container_selection> all </container_selection>
    <device_count> at_least_1 </device_count>
    <level> loose </level>
  </device>
</requirement>
```

Requirements can be either strict or loose. The motion sensor requirement is strict, which means that the app will not work without the motion sensors. But the contact sensor requirement is loose, which means that the app will work better with the contact sensors. However, it will still work without the contact sensors.

For specifying the positions of the sensors, DepSys uses a novel concept, **HomeTree**, where the position of an object is specified in a hierarchical fashion. The home is a container that contains all the rooms. Each room itself is a container containing all the objects within that room. The position NULL in the above example is because the app doesn't require placing the motion sensor in a particular position of the room.

There are cases where an app may need to specify the position of sensors. For example, in sleep monitoring in Empath[12], exactly 3 accelerometers need to be placed in the bed at certain positions: two are at the left and right of the middle of the bed and one is at the middle of the top of the bed. To specify such positions and to require at least 1 Hz sampling rate of the accelerometers, an app may specify its requirement as follows:

```
<requirement>
  <device>
    <device_type> Tri_axis_accelerometer </device_type>
    <position> Left_of_middle, right_of_middle,
               middle_of_top </position>
    <container> bed </container>
    <container_selection> any </container_selection>
    <device_count> 3 </device_count>
    <sampling_rate> 1 Hz </sampling_rate>
    <level> strict </level>
  </device>
</requirement>
```

Here, *container_selection=any* means that the app will work if there is any bed that satisfies the required sensor deployment. However, if the app wants to ensure that the bed has to be in the bedroom, and the bed has to be the master bed, it can specify:

```
<container_type> master_bed </container_type>
<container_room_type> bedroom </container_room_type>
```

After placing the sensors, the deployers specify the HomeTree of the deployment. Requirement Dependency Checker (RDC) in DepSys traverses the HomeTree and checks compatibility of sensor deployment with the requirements of the app. If the deployed sensors satisfy the requirements, the app can be installed and run in that home. Runtime dependencies are discussed later.

4.2 Addressing Name Dependency

We have two separate design choices to address name dependencies (c.f. Section 4). Either we create a standard set of terminologies to describe app requirements and resource availability and require all the app developers and deployers to use the same set of terminologies, or we let the app developers and deployers choose their own terminologies and use some machine learning techniques to infer if two keywords have the same meaning. We choose the former option as misclassification may lead to inaccurate conflict resolution that may threaten the life of the residents. However, we do realize that an app developer may misspell a terminology, or may forget to specify a requirement. DepSys is smart enough to detect such a "Missing Dependency" by monitoring the runtime behavior of the app (c.f. Section 5.3).

4.3 Addressing Sensor Control Dependency

There are sensors for which no control dependency usually arise and multiple apps can share them if needed, e.g., X10 motion sensors, weight scale, and contact sensors. However, for some sensors, multiple apps may want to use them in different ways. For example, two apps may want to use the same accelerometer at a different rate. State of the art solutions lack the ability to consider all the three dimensions: app priority, sensor resource availability constraint, and app requirements. For example, Android [1] allows apps to choose a rate from a set of only 4 available rates. PhysicalNet [19] uses

priority in resolving rates, but it doesn't consider sensor resource availability constraints, and rate selection strategy is not clearly specified in the paper. DepSys offers a novel **priority and resource availability constraint aware rate adjustment** strategy for resolving such dependency at runtime by considering all the aforementioned 3 dimensions (c.f. Algorithm 1). Although Algorithm 1 is simple, we describe it to illustrative the type of resolution that is needed.

Assume that n apps try to access accelerometer *AC1* at rates r_1, r_2, r_3, ..., r_n with priorities p_1, p_2, p_3, ..., p_n. Priorities are selected before runtime (c.f. Section 5.2). *AC1* may have a constraint on energy, e.g., the user/deployer may configure that it has to last for 30 days with the current battery. Assume that the maximum rate it can satisfy with its energy budget is r_max. Sensor Control Dependency Resolver (SCDR) in DepSys-Runtime Check (DepSys-RC) uses Algorithm 1 to resolve rates.

Algorithm 1 : ResolveRate(r_max, $rates$=[r_1, r_2, r_3, ..., r_n], $priorities$=[p_1, p_2, p_3, ..., p_n].)

```
1:  selected_rate ← r_max
2:  S ← sort(rates)
3:  for i ← 1 to n do
4:      g ← GCD(S(i), S(i + 1), ..., S(n))
5:      if g < selected_rate then
6:          selected_rate ← g
7:          break
8:      end if
9:  end for
10: for i ← 1 to n do
11:     rr_i ← floor(S(i)/selected_rate) * selected_rate
12: end for
13: resolved_rates ← sort([rr_1, rr_2, rr_3, ..., rr_n])
14: return resolved_rates
```

When an app asks for a particular rate, Algorithm 1 tries to provide the closest rate possible by considering all the higher priority apps and the constraints on resources. Line 2 of algorithm 1 sorts rates based on the app priority (rate of the lowest priority app comes first in S). If GCD of all the rates g is smaller than r_max, then we sample the sensor at a rate g and satisfy all the apps. Otherwise, we ignore the rate of the lowest priority app and try to satisfy the remaining higher priority apps. At line 13 of algorithm 1, we sort the rates in a way that the rate of the ith app appears at the ith index of $resolved\_rates$. Algorithm 1 returns the closest rate of each app that DepSys can support.

Sensors may have bandwidth constraints as well. For example, the above acoustic sensor *AC1* may have a maximum allowed bandwidth of 20 kbps that limits its maximum rate to r_max2. In that case, DepSys considers the minimum of r_max and r_max2 in line 1 of Algorithm 1, because that is the maximum allowed rate of *AC1*.

4.4 Addressing Actuator Control Dependency

Control dependency of the actuators is different from that of sensors and resolving it in a wrong way may cause user dissatisfaction, e.g., turning off light when it should be on, or may cause even death, e.g., granting an app's request to turn off the breathing machine to save energy while it is being used by another health app. State of the art solutions, e.g., HomeOS [13] detect such conflicts when two apps try to access the same device at the same time and resolve the conflict in favor of the higher priority app. DepSys goes beyond detecting such conflicts within a device, as it can detect conflicts across devices and differentiates false conflicts from true conflicts by considering the impact of the device on the environment and device semantics by using novel *effect*, *emphasis*, and *condition*. DepSys requires app developers to specify in XML *effect*, *emphasis*, and *condition for each actuator* that the app wants to control.

4.4.1 *Effect*

Effect specifies the effect of an app on the environment when using a particular device. Two apps may be using completely different devices, but are conflicting with each other by causing opposite effects. For example, *App1* may want to run humidifier while *App2* is running dehumidifier. Specifying *effect* enables the detection of such conflicts. The *effect* of a device is specified in the device driver by the driver developers. App developers may specify *effect* when the app's *effect* is a subset of the device's *effect* specified in the device driver. If app developers do not specify any *effect*, then DepSys considers all *effects* specified in the device driver, which can lead to pessimistic conflict detection.

We need to make sure all the app developers and device driver developers use the same terminology for specifying the *effect* of their apps. Based on the environmental conditions that affect human comfort [10], we propose to use <temperature>, <radiant_temperature>, <humidity>, <air_motion>, <odor>, <dust>, <aesthetics>, <acoustic>, and <light> XML tags. We differentiate playing a beep, like an alarm, from playing music or a continuous sound and suggest to use <beep> when a small alert will be generated and to use <acoustic> otherwise.

The text content of these XML tags can be *increase*, *decrease*, and *change*. For example, *App1* may specify:

```
<device>
   <device_name> humidifier </device_name>
   <effect>
     <humidity> increase </humidity>
   </effect>
</device>
```

On the other hand, *App2* may specify:

```
<device>
  <device_name> dehumidifier </device_name>
  <effect>
    <humidity> decrease </humidity>
  </effect>
</device>
```

This XML tag list and the text content of the XML tags are extensible. By comparing *effect*s of two apps, DepSys classifies them into one of four categories: (1) *same effect*, e.g., two apps want to increase light intensity, (2) *opposite effect*, e.g., the humidifier and dehumidifier case stated above, (3) *mixed effect*, e.g., two apps want to change room temperature in different ways, and (4) *different effect*, e.g., one app wants to increase sound while another app wants to increase temperature.

When two apps have *different effects*, they will not conflict. When two apps have an *opposite effect*, there is a chance that they will conflict at runtime. When two apps have the *same effect*, or *mixed effect*, DepSys looks into *emphasis* and *condition* to determine whether these two apps will conflict or not.

4.4.2 *Emphasis*

Emphasis is based on the **insight** that *not all control operations are equally important to an app,* and *emphasis* allows an app to specify which device operation is more important than others. Recall the conflict example of the security app and the energy app in Section 2 that demonstrates the limitation of the state of the art solutions that just use priority to resolve conflicts. For this particular example, it is important for the security app to turn on light $L1$ at 8 PM, but not so important to turn off light at 9 PM. On the other hand, for the energy app, it is important to both turn on and turn off lights. If both apps specify their *emphasis* in the meta-data and the security app is set higher priority than the energy app, then $L1$ is not turned off at 9 PM as long as the energy app wants to keep it on. Thus *emphasis* allows DepSys to differentiate false conflicts (conflict at 9 PM) from true conflicts (conflict at 8 PM) and resolve accordingly. True conflicts and false conflicts depend on device semantics and more examples of these two types of conflicts are specified in Section 5.3.

App developers specify *emphasis* in the XML meta-data of each app *for each actuator/device* it wants to control. Similar to *effect*, we need to make sure all the apps use the same terminology for specifying *emphasis*. Allowed terminologies are based on the allowed operations specified in the device drivers.

The security app mentioned above specifies its *emphasis* for controlling lights:

```
<emphasis>
  <operation> On() </operation>
</emphasis>
```

The energy app mentioned above specifies its *emphasis* for controlling lights:

```
<emphasis>
  <operation> On() </operation>
  <operation> Off() </operation>
</emphasis>
```

An app that wants to increase room temperature by using HVAC after detecting depression may specify:

```
<emphasis>
  <operation> On() </operation>
  <operation> ChangeSetpoint(72) </operation>
</emphasis>
```

72 degrees F is the target setpoint temperature. If the target setpoint is variable, then the app can use the VARIABLE keyword, as shown below:

```
<operation> ChangeSetpoint(VARIABLE) </operation>
```

When two apps have the *same emphasis*, then they are not conflicting with each other, e.g., if two apps' *emphasis* is to turn on light *L1*, then we can turn on *L1* and satisfy both of them. However, if the *emphasis* of two apps is *different*, e.g., App1's *emphasis* is to turn on *L1* while App2's *emphasis* is to turn off *L1*, then they may be conflicting (also depends on *conditions*). There may be other cases when two apps' *emphases* are *different*, e.g., when both apps' *emphasis* is to both turn on and off, or both apps' *emphasis* is ChangeSetpoint(VARIABLE), or App1's *emphasis* is ChangeSetpoint(arg1) while App2's *emphasis* is ChangeSetpoint(arg2) and |arg1 - arg2| > T, where T is a threshold specified in the device driver by the device driver developers by considering device resolution and the impact of the threshold in the perception of the user. It may be configured by the user.

4.4.3 *Condition*

Two apps are not conflicting if they operate on a device with a mutually exclusive condition. App developers specify *condition* in the XML meta-data of each app *for each actuator/device* it wants to control. Apps may actuate on devices on a variety of conditions and the conditions can be categorized into two groups: (1) conditions based on time, e.g., at sunrise, sunset or at 9:00 PM and (2) conditions based on events, e.g., conditions based on (a) actuators, e.g., when the front door is open, (b) sensors, e.g., when a motion sensor in the living room fires, (c) activities of daily living, e.g., when a resident is eating or sleeping, (d) environmental state, e.g., when there is a flood, fire, or earthquake, or (e) physiological and psychological status of the residents, e.g., when someone is depressed or having insomnia. We only take into account *conditions* based on time for two reasons. First, the goal of using *condition* is to determine whether the *conditions* specified by two apps on a particular device are mutually exclusive or not during installation time and *conditions* based on events are usually not mutually exclusive. For example, almost all the aforementioned events can take place at the same time, although the probability is low. Second, even if app developers specify such a condition, e.g., *App1* turns on lights in the living room when the resident is depressed, DepSys can not verify whether *App1* is obeying such *condition* at runtime as DepSys doesn't know the internal logic of any app.

To specify *condition*, app developers use the XML tags: start_time, end_time, all_time, night, day, sunrise, sunset, dawn, and dusk. The options for text contents are HH:MM:SS in 24 hour format, begin, end, any, and duration. Here are some examples:

```
<start_time> 20:00:00 </start_time>
<end_time> 21:00:00 </end_time>
<night> duration </night>
<sunset> begin </sunset>
```

The first two lines specify the exact time of operation (between 8PM to 9PM), the third one specifies that the device will be used any time during the night, and the fourth one specifies that the device will be used when sunset begins. The Temporal Translation Engine (TTE) module in DepSys converts temporal events, e.g., sunset or sunrise to time of day by using the location and year-long environmental information. The *conditions* are compared for determining conflicting apps during installation time in the DepSys-Static Check module.

5. CONFLICT DETECTION, RESOLUTION, AND USER ROLE

In this section, we summarize the conflict detection and resolution strategies of DepSys and the role of the user for specifying policy.

5.1 Dependency Check at Installation Time

Assume that the user has already installed N apps: $App1, App2, App3, ..., AppN$. When he tries to install a new app $AppM$, DepSys-SC checks for requirement dependencies (c.f. Section 4.1), and if the deployment satisfies app requirement, it performs a dependency check between $AppM$ and all the N previously installed apps for actuator control dependency, one pair at a time. Note that sensor control dependency is addressed and resolved automatically only at runtime (c.f. Section 4.3). Let's say DepSys-SC is performing actuator control dependency checking between $AppM$ and $Appi$ ($1 \leq i \leq N$) using their XML meta-data. If the two apps want to use multiple devices, then DepSys-SC does the following check for every pair of devices, where each pair consists of 1 device from $AppM$ and 1 device from $Appi$.

If $AppM$ and $Appi$ are using different devices, then there is no need to see the *emphasis* of the two apps. In this case, the truth table for detecting dependency conflict is shown in Table 1. It shows that if the *conditions* are mutually exclusive, or the *effect* is *different* or *same*, then these two apps are not conflicting. Otherwise, there is a potential chance of conflict between these two apps.

If $AppM$ and $Appi$ are using same device, then *emphasis* is used in the dependency checking. Then the truth table for detecting a dependency conflict becomes like Table 2. It shows that if the *conditions* are mutually exclusive, or the apps have the *same emphasis*, or the apps have *different effects*, then they are not conflicting. Otherwise, they may be conflicting.

Effect	*Condition*	Conflicting?
-	Mutually Exclusive	No
Same	Not Mutually Exclusive	No
Opposite	Not Mutually Exclusive	Yes
Mixed	Not Mutually Exclusive	Yes
Different	-	No

Table 1: Truth table for conflict detection

Effect	*Emphasis*	*Condition*	Conflicting?
-	-	Mutually Exclusive	No
-	*Same*	-	No
Different	-	-	No
Same	*Different*	Not Mutually Exclusive	Yes
Opposite	*Different*	Not Mutually Exclusive	Yes
Mixed	*Different*	Not Mutually Exclusive	Yes

Table 2: Truth table for conflict detection

Let's say, DepSys-SC detects that $AppM$ is conflicting with j previously installed apps $App1, App2, App3, ..., Appj$. To learn the policy of conflict resolution if these apps conflict at runtime, the Preference Learning Module (PLM) takes input from the user.

5.2 Role of the user

At the app installation time, when it is determined that a particular app $AppM$ may conflict with other apps (c.f. Section 5.1), HomeOS [13] requires the user to specify app priority and maintain a total order among the conflicting apps. The Preference Learning Module (PLM) of DepSys offers a novel solution called Semantic Aware Multilevel Equivalence Class based Policy (SAMECP) that reduces the cognitive burden of the users and allows apps to be run in a more flexible way.

SAMECP categorizes the apps into four groups: energy, health, security, and entertainment. App developers specify the group in which their app belongs. By default, SAMECP maintains a priority across groups so that health > security > entertainment > energy. However, the user can change the priority of groups if needed. If $AppM$ belongs to health and the other conflicting apps belong to energy or entertainment group, then although they are conflicting, their priority is already established and no user feedback is needed, thus reducing the cognitive burden.

However, priority needs to be determined between two apps when they belong to the same group. SAMECP uses an interesting **insight** in this case, which is, *for a number of apps installed in a home, it really doesn't matter which app takes control as long as it does not break the semantics of appliance usage.* For example, assume that there are 3 apps installed for playing music. $App1$ plays music based on heart rate, $App2$ plays music based on the words in the residents' speech, and $App3$ plays music based on the environment (e.g., rainy, cloudy, and snowy). It may not matter which app plays

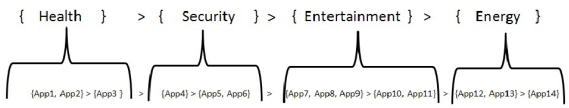

Figure 2: Semantic Aware Multilevel Equivalence Class based Policy

music at a particular moment as long as they are not fighting and the music is not turned on and off over and over again. This brings up the idea of using *equivalence classes of priorities*, where instead of prioritizing each app by comparing with all other apps within the same group, a number of equivalence classes of priorities are created and the user puts an app into its class. It is called multilevel equivalence class based solution, because at a higher level there are 4 groups and in a lower level there are equivalence classes of apps as shown in Figure 2. There are priorities assigned for each equivalence class, but within a class, apps are not prioritized and the user specifies the policy about how to decide which app to run. For this particular example, the user puts all three apps into a single class and specifies the policy of selecting these apps, e.g., random selection, or preference probabilities (60% time *App*1, 40% time *App*2 and *App*3). Such policies increase the flexibility of the ways apps are run.

5.3 Runtime Check and Addressing Missing Dependency

In this section, we describe the runtime conflict detection and resolution of DepSys by the DepSys-Runtime Check (DepSys-RC) module. At runtime, when an app wants to control a device, the request has to go through DepSys-RC. Sensor Control Dependency Resolver (SCDR) in DepSys-RC resolves the control dependencies of sensors automatically as described in Section 4.3.

When an app wants to control an actuator at runtime, the device control request contains *effect*, *emphasis*, and *condition*. DepSys-RC uses *effect* and *emphasis* to categorize conflicts into **true conflicts** and **false conflicts**. For example, when an app wants to turn on a light that another app wants to turn off, that is a true conflict. Similarly, when an app wants to keep a light on while another app wants to change its light intensity, that is also a true conflict. However, from table 3, when *App*#1 wants to turn off light at 9PM, this is an optional off request. If another app, say *App*#7 wants to keep it on at that time, then the conflict at 9 PM is actually a false conflict.

For HVAC, when an app wants to turn on the HVAC controller while another app wants to turn it off or wants to run with a different setpoint, that is a true conflict. However, when an app wants to turn on an HVAC controller while another app wants to turn it off optionally, e.g., turn on of *App*#7 and optional turn off of *App*#17 from table 3, this is a false conflict. Note that true conflicts can happen across devices, e.g., one app is running a heater and another app is running an AC. Such conflicts are detected using *effect* and not by any sensor. So, even if there is a time gap between actual conflicts, e.g., if the heater takes some time to start heating, we can still detect such a conflict by comparing the *effect* of each actuation. When there is a true conflict, the Actuator Control Dependency Resolver (ACDR) in DepSys-RC takes into account the policy and priority set by the user in the Preference Learning Module (PLM) and arbitrates actuator/device access accordingly. However, if there is a false conflict, DepSys just ignores the optional request regardless of app priority.

App developers may forget to specify a sensor/actuator dependency or requirement, or even misspell the terminology. But when the app tries to access the sensors and actuators at runtime, such missing dependencies are detected by the Missing Dependency Checker (MDC) in DepSys-RC. When *App*1 asks to turn on *L*1, the device control request contains *effect*, *emphasis*, and *condition*. MDC can detect two types of missing dependencies by comparing the device control request with the XML metadata that the app provided during the installation time: (1) missing requirement dependency, e.g., *App1* did not specify its requirement to use *L1* during installation time, but at runtime *App1* is trying to control *L1*, and (2) missing control dependency of the actuator, e.g., missing *effect*, *emphasis*, and *condition*.

When a missing dependency is detected, DepSys updates the app's dependency information assuming the runtime dependency description is accurate and runs a dependency check by DepSys-SC across all other installed apps. If a conflict is detected, it may need to get user feedback to update the policy (c.f. Section 5.2).

6. EVALUATION

Although DepSys addresses a spectrum of dependencies, its main novelty and effectiveness lies in detecting and resolving control conflicts of sensors and actuators in a home setting. Among the sensor and actuator control dependencies, actuator control dependency detection and resolution is relatively more difficult. Since Algorithm 1 resolves sensor control dependency in a superior way than state of the art solutions by considering all the three dimensions (c.f. Section 4.3), we limit the evaluation of DepSys in detecting and resolving actuator control dependencies.

To evaluate DepSys, we need an app store for the home. Although there are popular app stores for smart phones, e.g., Apple app store[4] and Google play[6], the apps in these app stores are mainly limited to smart phones and tablets. To the best of our knowledge, there is no such well-established app store for the home. Hence,

ID#	App Name	Category	App Description
1	Discourage Burglar	Security	It turns on all the lights at 8 PM and turns off all the lights at 9 PM.
2	Door open alert	Security	It turns on all the lights and plays an alert sound in all the speakers when the front door is open for more than 2 minutes. It keeps the lights on and alert playing until the door is closed.
3	Door and window open notification	Security	When any door or window is opened, it just plays a 10 second beep sound in the speaker.
4	Sleep time Door and Window Protection	Security	It closes all the windows and doors when someone goes to sleep and keeps them closed until he wakes up.
5	Suspicious activity Reporter	Security	It turns on all the lights and plays a beep sound in the speaker when suspicious activity is detected by the security cameras.
6	Smoke alarm	Safety	It turns on all the lights and plays fire alarm in the fire alarm device when smoke is detected.
7	Home Energy Control	Energy	It turns on HVAC and light of the occupied rooms based on motion detection. It turns off light, HVAC after 10 minutes and 30 minutes of no motion detection, respectively.
8	Bedroom TV Management	Energy	It uses accelerometers in the bed to detect if someone is falling asleep. When that happens, it turns off the TV in the bedroom.
9	Kitchen Energy Management	Energy	It plays a beep sound for 30 seconds if the stove is on for unusual period of time to make sure someone didn't forget to turn it off.
10	Smart HVAC	Energy	It turns on HVAC by monitoring the GPS coordinates of the residents, e.g., when someone is coming towards home, it turns on HVAC when he is within 15 miles of the home.
11	Humidifier Control	Energy	It turns on humidifier when humidity drops below a threshold.
12	Dehumidifier Control	Energy	It turns on dehumidifier when humidity exceeds a threshold.
13	Budget based HVAC	Energy	It turns on, turns off HVAC in a way that meets daily energy budget for the HVAC system.
14	Sunset	Energy	It turns on lights in all the rooms for 5 minutes when the sun is set.
15	Light control during sleep	Energy	It uses accelerometers in the bed to detect if someone is falling asleep. When it happens, it turns off lights of the bedroom. It keeps the lights off while sleeping and turns them on when he wakes up from bed.
16	Light Mode	Energy	The app offers different modes of light control. For example, while watching a movie, the residents' can choose a 'movie mode' that lowers light intensity. Other mode options are 'party mode', 'candle light dinner mode' etc. The residents need to select the mode by themselves.
17	Activity based HVAC control	Energy	It monitors Activities of Daily Living (ADLs) and controls HVAC accordingly. For example, if someone is preparing a meal or eating, it reduces the setpoint of the kitchen by 1 degree F. When someone is sleeping at night, it increases the bedroom setpoint temperature by 1 degree F at the last hour of the sleep.
18	Mood assistance	Health	When a depression episode is detected, it turns on lights in the occupied rooms. It keeps the lights on until the resident goes to sleep or the depression status is improved. It also increases room temperature.
19	Seasonal Affective Disorder Control	Health	It makes sure that lights in the bedroom are not turned off before 10 PM. It also makes sure that the lights are turned on no later than 7 AM.
20	Med reminder	Health	It makes a beep sound in the speaker when it is the time to take medication.
21	Food control	Health	It flashes light intensity in the kitchen/dining room and plays a beep sound at the nearest speaker for 1 minute if dining activity exceeds more than an hour.
22	Pollen control	Health	It keeps the windows closed when there is pollen alert in that area.
23	Wind Blower	Weather	When it is windy outside, it opens all the windows in the occupied rooms.
24	Charm of rain	Weather	It flashes light intensity in the occupied rooms and plays thunderstorm sound in the speaker when it rains
25	Weather Alert	Weather	When someone opens the front door, it plays a beep sound in the speaker if there is a rain or thunderstorm forecast in that day.
26	Alarm clock	Alert	It turns on lights of the occupied rooms and plays a beep sound in the speaker for a minute at timeout.
27	Calendar	Alert	It turns on lights of the occupied rooms and plays a beep sound in the speaker 10 minutes prior to start of an event specified in Google calendar.
28	Social Networking	Alert	It plays a beep sound in the speaker when messages are received from friends or supervisors in Facebook and Gmail.
29	Musical Heart	Music	It plays music in the speaker based on heart rate of the residents.
30	Music of environment	Music	It plays music in the speaker based on weather conditions, e.g., cloudy, windy, snowfall, rain, shower, thunderstorm, lightening etc. It also changes light intensity of the occupied rooms to show similar effect.
31	Music for activities	Music	It plays music in the speaker based on activities of daily living, e.g., entering home, preparing meal, and eating.
32	Basketball	Game	When this app is played in the computer, it uses the speaker and the lights of the room where the computer is placed.
33	Baseball	Game	When this app is played in the computer, it uses the speaker and the lights of the room where the computer is placed.
34	Need for Speed	Game	When this app is played in the computer, it uses the speaker and the lights of the room where the computer is placed.
35	Quake	Game	When this app is played in the computer, it uses the speaker and the lights of the room where the computer is placed.

Table 3: Our App Store containing 35 apps

we create a number of apps from various categories, including energy, health, security, and entertainment that will serve as our app store. Our app store contains 35 apps as shown in Table 3. The way we create these apps is by designing them and defining the metadata without the real implementation. The apps are very representative of those from papers in literature and app stores for smart phones. As some of the apps share sensors and actuators, someone may question the selection of these apps. Note that it is our goal to allow apps to share sensors and actuators, and considering 1 million and 900,000 apps in Android's and Apple's app store, respectively [8], when we have similar number of apps in an app store for the home, it is not unreasonable to assume that some apps will share sensors and actuators. We evaluate DepSys's conflict detection at installation time and at runtime separately.

6.1 Static Analysis

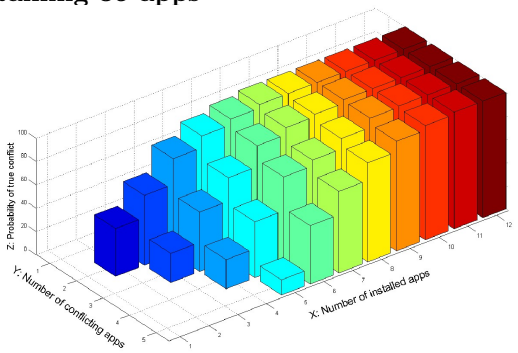

Figure 3: Static analysis of conflicts among apps

We assume that the sensor/actuator deployment in the home supports all the requirements of the 35 apps in Table 3. Before running an app, DepSys-SC performs static analysis at installation time by analyzing the dependency information specified in the app metadata. Figure 3 shows the probability of true conflict between at least j apps when i apps are installed from

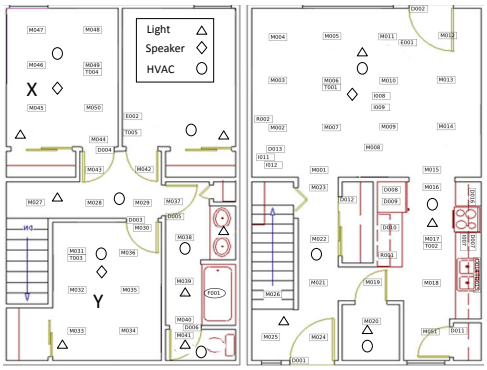

Figure 4: Floorplan and position of sensors and actuators

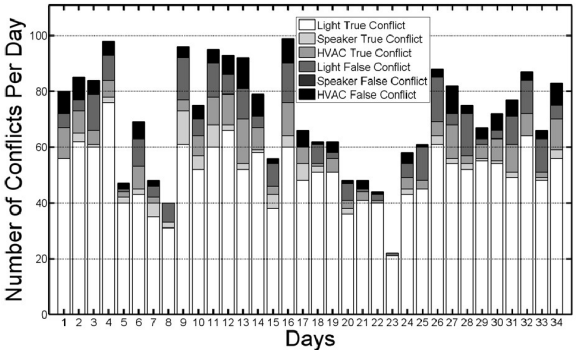

Figure 5: Number of conflicts at different days

the 35 apps in Table 3. For each value x of the X axis, we randomly select x apps from the 35 app list 100 times and compute the probability of true conflict between at least y apps, where $1 \leq y \leq 5$ and $y \leq x$, and show the average results in Figure 3. Figure 3 shows that when someone installs 2 apps from the 35 app list, there is a 41% probability that these apps will be conflicting. When someone installs 5 apps there is 92% probability that at least two apps will be conflicting. When someone installs 11 apps, there is a 100% probability that at least 5 apps will be conflicting. These results show the severity of conflicts among apps when multiple apps are installed in a home setting and demonstrate the need for detecting and resolving conflicts.

6.2 Runtime Analysis

We use a dataset collected from WSU CASAS smart home project [11] for performing runtime analysis. We use 34 days of data in our analysis (from November 04, 2010 to December 07, 2010). Two residents were living in this home in two separate rooms (marked as X and Y). There were 11 lights. We assume that the home has 3 speakers (one in each individual's room and one in the living room) and 10 HVAC controllers. The floorplan and the position of the sensors and actuators are shown in Figure 4.

The actual deployment did not have any apps installed. However, the dataset provides enough information in terms of sensor data and ground truth of activities of the residents to determine the behavior of the following 10 apps (App# 1, 2, 3, 7, 14, 15, 17, 19, 21, and 31) in terms of how these apps will control the lights, HVAC, and speakers of the testbed. For example, we know that there were 11 lights in the testbed and App# 1 will turn on all of these at 8:00 PM and turn them off at 9:00 PM. We use the location of the testbed and the date of data collection to determine the sunset time and thus compute when lights will be on at sunset by App# 14. We compute when App# 7 turns on/off HVAC controllers based on motion sensor data. The dataset contains labeled ground truth of activities, e.g., when the two residents were eating, preparing meal, entering home, and sleeping. We use this ground truth of activities to determine when App# 15, 17, 19, 21, and 31 control lights, HVAC controllers, and speakers. As no windows were instrumented, we use only door contact sensors firing to determine when App# 2 and #3 control lights and speakers. The reason for selecting these 10 apps is that we are trying to determine the behavior of as many apps as possible from our 35 apps and the dataset allows us to determine the behavior of only these apps.

6.2.1 *Number of run-time conflicts*

Figure 5 shows the number of conflicts on different days when these 10 apps are installed. Each day's number of conflicts is broken down between conflicts in lights, speakers, and HVAC. The true and false conflicts of these devices are also shown. We see from Figure 5 that more conflicts occur with the lights than with the speakers and HVAC. No false conflicts are detected on the speakers. On average 70.71 conflicts take place per day out of which 11.74 conflicts are false conflicts, which is 16.60% of the total conflicts.

The number of conflicts depends on the number of apps installed. We vary the number of installed apps from 1 to 10. We take 100 samples from the 10 app list for each value of the X axis and find out the number of conflicts per day by using the CASAS dataset and present the average results in Figure 6. The magnitude of conflicts we see in Figure 6 is really surprising and it clearly shows that when people install more and more apps, the number of conflicts rises exponentially. It also shows that 16.54% conflicts remain false conflicts on average, which can be a significant number when people install hundreds of apps.

6.2.2 *Conflict resolution capability*

In this section, we compare the conflict resolution capability of DepSys with that of a state of the art solution, HomeOS [13]. DepSys is more powerful in resolving conflicts because it considers device seman-

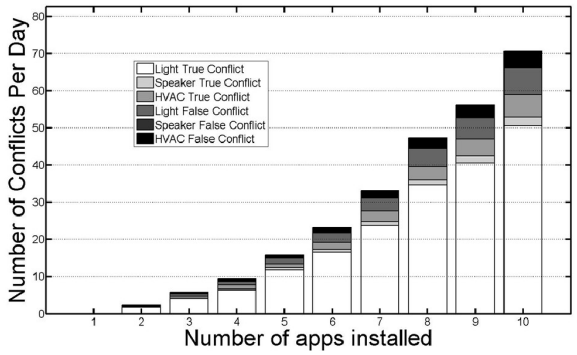

Figure 6: Number of conflicts for various number of apps installed

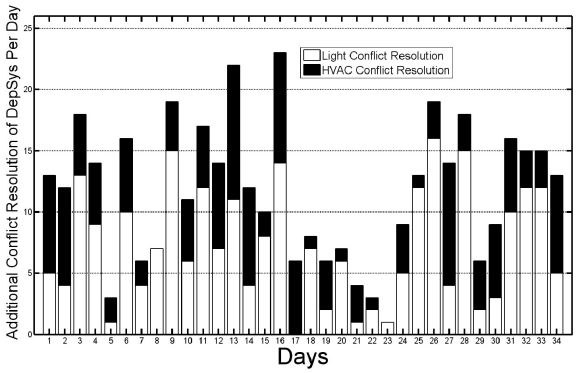

Figure 7: Additional conflict resolution of DepSys over HomeOS at different days

tics and can separate false conflicts from true conflicts (c.f. Section 5.3). When two apps conflict at runtime, HomeOS uses priority to resolve the conflict in favor of the higher priority app. If the conflict is a true conflict and HomeOS can resolve it accurately, DepSys can also do the same, as users can specify a similar policy in DepSys. However, if the conflict is a false conflict, HomeOS can't recognize it and resolves the conflict in favor of the higher priority app. But DepSys uses *effect* and *emphasis* to determine whether this operation is important or optional for the higher priority app (c.f. Section 5.3). If the operation is optional, DepSys ignores the request of the higher priority app and resolves it in favor of the lower priority app. We compute the number of times a priority based system like HomeOS fails to recognize and resolve such conflicts and as DepSys can resolve such conflicts accurately, we call such events *additional conflict resolutions* of DepSys.

To compute the number of additional conflict resolutions of DepSys per day, we use CASAS dataset and use the 10 apps (App# 1, 2, 3, 7, 14, 15, 17, 19, 21, and 31). We assume that security apps have highest priority, followed by health, entertainment, and energy apps. More specifically, the priorities of the 10 apps are 1, 2, 3, 10, 9, 4, 8, 5, 6, and 7, respectively, 1 being the highest priority and 10 being the lowest priority.

Figure 7 shows the number of additional conflict resolutions of DepSys per day for this priority scheme for the 10 apps. The result is broken down into lights and HVAC. We do not observe any such conflicts in speakers. Although the number of additional conflict resolutions of DepSys per day is 11.74 on average, it can be as high as 23 in a day, some of which can be potential discomfortable events. These results show the conflict resolution capability of DepSys over HomeOS, as none of such conflicts can be resolved accurately by HomeOS.

6.2.3 App parameter selection

DepSys monitors conflicts among apps and computes a level of conflict for each app, which is the number of times an app experiences true conflict with other apps per day. If the level of conflict is higher for an app, it means the app is more conflicting. If the level of conflict of an app is high, it can be used to suggest changing some parameters of the app. For example, App# 7 (Home Energy Control) uses a 10 minute timeout interval before turning off lights, for which it's level of conflict is 55.26. We change the timeout interval of App# 7 from 5 minutes to 90 minutes and show how it affects the level of conflict of each app in Figure 8. We see that the level of conflict of App# 7 decreases from 75.44 to 21.71 when the timeout interval is changed from 5 minutes to 90 minutes. As Figure 8 shows, the level of conflict can be used to choose appropriate parameters of an app so that the level of conflict of an app remains within a bound. It can also be used to detect and isolate the most conflicting apps.

7. DISCUSSION

In this work, we advance the state of the art techniques to specify, detect, and resolve a wide range of conflicts due to dependencies in a smart home. However, DepSys's improved capabilities in conflict detection and resolution comes at the cost of developers' efforts in specifying additional dependency information. However, the effort to specify additional dependency information containing *effect*, *emphasis*, and *condition* is minimal as the average number of lines per app is 25.17, considering the 35 apps specified in Table 3. Note that DepSys also requires addressing requirements and name dependencies, but the developers need to put similar effort if they use other state of the art solutions, e.g., HomeOS and hence these are not considered an additional effort for specifying dependency.

There are some limitations in this work. For example, DepSys is designed for non-safety critical systems. The residents can jeopardize their health by choosing an erroneous policy, e.g., assigning a higher priority to an energy management app over a health care app that controls the breathing machine. Also, *effect* only considers

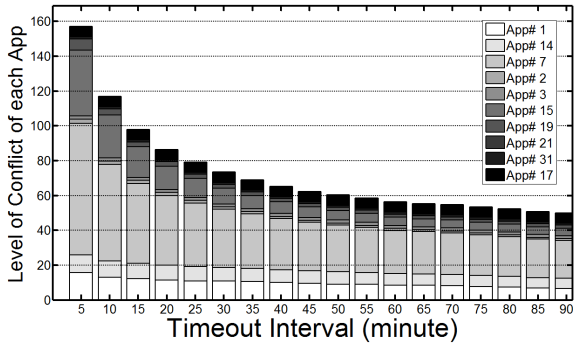

Figure 8: Level of conflict for different timeout intervals of App# 7

the environmental factors that affect human comfort. In the future, we will consider other impacts, e.g., impacts on the human body due to medicine or food. That will enable detecting redundant apps, e.g., a diabetic patient may install two apps for monitoring glucose level and administering insulin. As both apps administer insulin independently, there could be an overdose. Although, the use of *effect* allows us to detect such issues, it is not easy. Multiple apps may administer multiple drugs that may increase each others' effects (e.g., using antihistamines with alcohol) or may decrease each others' effects (e.g., using alcohol with caffeine) in the human body. Some of these effects could be acceptable and intentional, e.g., to reduce self-consciousness, it may be acceptable to take antihistamines with alcohol.

Although DepSys is designed for smart homes, some of its principles can be generalized to other application domains, e.g., industrial process control. An app based paradigm can be applied to an industrial process control, where each control loop can be treated as an app. Multiple apps, i.e., control loops may conflict on a single actuator or multiple actuators depending on the *effect* and *emphasis* of their actuation. The metadata for specifying the *effect* and *emphasis* needs to be determined from the application context. An industrial process control system is usually a closed system, where modules are usually developed by the same group to work together. Also, it is usually a rigid system as dependency checking can be performed during offline static analysis assuming that components will not be added and removed dynamically. On the other hand, DepSys is an open system, where different apps are developed by different external app developers without knowing each others assumptions and offers dependency checking of a much more flexible system where apps can be added and removed dynamically at runtime. CPS systems that exhibit such characteristics will benefit from our solution.

8. CONCLUSIONS

Cyber-Physical Systems in smart homes pose new challenges for sharing and controlling home physical world entities due to various types of dependencies. DepSys aims to address some of these challenges by providing comprehensive strategies to specify, detect, and resolve conflicts due to such dependencies. Its design offers an app based utility sensing and actuation infrastructure to run many CPS apps in the home. If app developers build their apps by specifying dependency information that DepSys requires, it will enable detecting and resolving a significant amount of conflicts in a home setting. In the future a full implementation of DepSys is planned and could be built upon a platform such as HomeOS.

9. ACKNOWLEDGEMENT

This paper has been supported, in part, by NSF grants CNS-1239483, CNS-1319302, and EFRI-SEED 1038271.

10. REFERENCES

[1] Android. http://www.android.com/.
[2] Leviton Online Store. http://www.levitonproducts.com.
[3] M1 Security and Automation Controls. http://www.elkproducts.com/m1_controls.html.
[4] Apple app store. http://www.apple.com/osx/apps/app-store.html.
[5] Control4 Home Automation and Control. http://www.control4.com.
[6] Google play. http://play.google.com/store?hl=en.
[7] Home Automation Systems. HomeSeer. http://www.homeseer.com.
[8] http://www.phonearena.com/news/Androids-Google-Play-beats-App-Store-with-over-1-million-apps-now-officially-largest_id45680.
[9] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys*, 2003.
[10] V. Bradshaw. *The Building Environment: Active and Passive Control Systems*. John Wiley & Sons, Inc., River Street, NJ, USA, 2006.
[11] D. Cook and M. Schmitter-Edgecombe. Assessing the quality of activities in a smart environment. *Methods of Information in Medicine*, 2009.
[12] R. F. Dickerson, E. I. Gorlin, and J. A. Stankovic. Empath: a continuous remote emotional health monitoring system for depressive illness. In *WH*, 2011.
[13] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *NSDI*, 2012.
[14] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *ICDCS*, 2005.
[15] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: a flexible and adaptive framework sensor networks. In *EWSN*, 2005.
[16] S. Nirjon, R. F. Dickerson, Q. Li, P. Asare, J. A. Stankovic, D. Hong, B. Zhang, X. Jiang, G. Shen, and F. Zhao. MusicalHeart: a hearty way of listening to music. In *SenSys*, 2012.
[17] L. Szumel, J. LeBrun, and J. D. Owens. Towards a mobile agent framework for sensor networks. In *EmNets*, 2005.
[18] P. A. Vicaire, E. Hoque, Z. Xie, and J. A. Stankovic. Bundle: a group based programming abstraction for cyber physical systems. In *ICCPS*, 2010.
[19] P. A. Vicaire, Z. Xie, E. Hoque, and J. A. Stankovic. Physicalnet: A generic framework for managing and programming across pervasive computing networks. In *RTAS*, 2010.
[20] A. Wood, J. Stankovic, G. Virone, L. Selavo, Z. He, Q. Cao, T. Doan, Y. Wu, L. Fang, and R. Stoleru. Context-Aware Wireless Sensor Networks for Assisted Living and Residential Monitoring. *IEEE Network*, 22(4):26–33, Jul./Aug. 2008.
[21] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys*, 2006.