Integration of Cyber-Physical Systems for Smart Homes

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Sirajum Munir

December 2014

Abstract

As sensor and actuator networks mature, they are becoming a core utility of smart homes like electricity and water. As electricity enables running many electrical appliances in the home, the sensor and actuator networks enable the running of many Cyber-Physical Systems (CPSs) from different domains including energy, health, security, and entertainment. It has been hypothesized that integrating these CPSs in a smart home will offer innumerable advantages including achievement of the positive synergistic effects of the CPSs and avoidance of the negative consequences. However, integrating these CPSs is very challenging. Because, each individual system has its own assumption and strategy to control the physical world entities without much knowledge of the other systems. As a result, when these systems are integrated in a home setting without careful consideration, they raise many systems of systems interdependency problems.

In this thesis, we propose a utility sensing and actuation infrastructure for smart homes called DepSys that integrates many CPSs in a home by treating each system as an app. From the application layer to the link layer, DepSys provides the most comprehensive strategies to address a spectrum of dependencies including dependencies on sensor and actuator level control, sensor reliability status, involving human-in-the-loop, and on real-time constraints for delivering packets within latency bounds over wireless networks. We are the first to demonstrate that dependency metadata that focuses on the effect on the environment enables us to detect conflicts across devices that isn't possible by monitoring actuations of individual devices as performed by the state of the art solutions. We also specify novel metadata that helps understanding context and resolving actuator level conflicts more accurately that couldn't be achieved through existing priority based solutions. We develop a novel sensor failure detection scheme called FailureSense that addresses not only fail-stop failures, but importantly, obstructed-view and moved-location failures; that are realistic and common in smart homes and barely addressed to date in the literature or in real deployments. Our solution is the first one to detect these failures without requiring sensor redundancy and with minimal training effort. To address human-in-the-loop dependency, we develop EyePhy that uses a simulator to model the complex interactions of the human physiology using over 7800 variables. Using the simulator, EyePhy determines the potential conflicts among the human-in-the-loop apps' interventions. It provides the most comprehensive dependency analysis to date by taking into account a wide range of physiological parameters and allows the dependency analysis to be personalized. Some smart home applications

may have dependencies on real-time constraints, where packets need to be delivered within latency bounds reliably over wireless networks. To address this need, we perform a 21 days long empirical study, where we transmit 3,600,000 packets over every link of a 802.15.4 testbed to characterize wireless links. Based on the findings, we design a static network-wide stream scheduling algorithm that uses a novel least-burst-route to produce latency bounds of real-time periodic streams by taking into account link bursts and interference. In addition to addressing these various types of dependencies, DepSys handles the case when app developers fail to specify dependencies. Although DepSys is designed for smart homes, some of the its principles can be applied to other CPS application domains, e.g., industrial process control.

APPROVAL SHEET

The dissertation

is submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

SIRAJUM MUNIR

AUTHOR

The dissertation has been read and approved by the examining committee:

John A. Stankovic

Advisor

Alfred C. Weaver

Gabriel Robins

Yanjun Qi

John Lach

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

December

2014

To my parents and wife

Acknowledgements

I would like to express my earnest gratitude to my superb advisor, Professor John A. Stankovic, who has helped me at every step of this research with his profound knowledge and endless patience. His insights on how to conduct research and passion for always improving the quality of a work have been the greatest source of inspiration and a remarkable learning experience in my research career.

I would like to thank my committee members Yanjun Qi, Alfred Weaver, Gabriel Robins, and John Lach for being very accessible and supportive to this work. They have always given me time despite of their busy schedules.

I can't find the right words to express my gratitude to my parents for their tremendous support throughout my whole life. They have taught me the importance of education from my early childhood and given me the best possible opportunities to study and learn to make my life full of promise.

I would like to thank my beloved wife, Sarah Masud Preum for bringing happiness and joy in my life. With her care, love, and humor, she has made staying abroad delightful.

Finally, I would like to thank all of my friends at Charlottesville who helped me and made my staying here easier. I feel extremely fortunate to be able to find so many good friends here and will cherish the wonderful memories I have with them for all the fun things we did here apart from the graduate study.

Contents

List of Tables

List of Figures

Chapter 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. Interferences arise from many issues, but primarily when the cyber depends on assumptions about the environment, the hardware platform, requirements, naming, control, sensor reliability, and human-in-the-loop. 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 thesis employs a utility sensing and actuation infrastructure for a smart home that allows running many CPS apps from many domains by addressing a spectrum of dependencies.

1.1 Motivation

It has been hypothesized that integrating Cyber-Physical Systems (CPSs) in a smart home will have synergistic effects. For example, let's assume that we integrate the CPSs 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. As an example, when the home health care system detects the occurrence of a depression episode, it can notify the energy management application and the energy management application can turn on the lights and increase room temperature. Alternately, the energy management system can help detecting sensor failure of the home health care application (shown below). Besides these positive synergistic effects, integration will allow avoiding 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. Also, instead of

using multiple instances of motion sensors in the same room, the energy management system, home health care system, and security system can share a single instance of a motion sensor. Sharing of sensors will reduce cost of deployment, improve aesthetics of the rooms, and reduce channel contention for packet transmission. The channel contention can be severe sometimes. As shown in [1], when X10 motion sensors are increased from 1 to 4 within the same radio cell, false data and node IDs begin to appear due to packet corruption and Packed Delivery Ratio (PDR) drops from 100% to 20%!

To address this need, we design a platform called DepSys for integrating many Cyber-Physical Systems in a smart home. We use an app based paradigm for integrating CPSs as DepSys treats each system as an app. Although there are commercial home automation and security systems [3] [4] [5] [6] are available on the market, 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. However, HomeOS [7] provides a PC-like abstraction for technologies in the home where external app developers can write their own apps. But, HomeOS lacks its capability in detecting and resolving control conflicts at the actuator level, detecting sensor failure, and detecting conflicts among apps' interventions due to human-in-the-loop that DepSys offers.

There are motivations for each of the dependencies that DepSys addresses. The motivations for a addressing several important dependencies are described below:

- Accurate resolution of control dependency of the actuators is important and resolving it in a wrong way may cause immediate user dissatisfaction, e.g., turning off a 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.

- If multiple systems in a smart home are integrated and allowed to share sensors, then a single sensor failure will affect the performance of all the dependent systems. Therefore, detecting sensor failure is crucial for the integrated system.

- Humans are intimately involved with the integrated smart home systems. Apps that perform interventions can increase or decrease each others effects on the physiological parameters of the involved human being due to human-in-the-loop dependencies. It is important to detect interventions that are conflicting or interacting, as some of which can be harmful to the user.

- Some smart home applications may have dependencies on real-time constraints for delivering packets wirelessly within latency bounds reliably. Also, we envision DepSys to be used in other areas in the future, e.g., in large scale industrial plants, where sensors and actuators will form a multi-hop ad hoc network with a real-time requirement of reliable packet delivery within latency bounds. To satisfy such requirements, we need to address dependencies on two physical properties of the wireless communication medium, link burstiness and interference.

1.2 Technical Challenges

Integrating multiple systems is very challenging as each individual system has its own assumption and strategy to control the physical world variables without much knowledge of the other systems. There are many dependency issues that need to be addressed in order to design a framework like DepSys. Each dependency poses one more more technical challenges. Some major technical challenges are listed below:

First, a smart home platform depends on the architecture of system integration. Over the years, a few architectures have been proposed for homes, including AlarmNet [8], Empath [9], and HomeOS [7]. There are strengths and weakness of each architecture. We need to choose an appropriate architecture for smart homes.

Second, we expect that app developers will specify dependency information and users will select policy for conflict resolution. Our platform DepSys will detect conflicts using the dependency information specified by the app developers and resolve conflicts by following the user defined policy. We do not want to burden the app developers and users by asking to specify a lot of information. Therefore, a major challenge is to figure out how to distribute responsibilities between app developers and users so that app developers have to exercise minimal effort in specifying dependency, users have less cognitive burden in specifying policy, yet DepSys can detect and resolve a wide range of conflicts across apps, and apps can be run in a more flexible way than the state of the art solutions. We also need to consider the case when app developers forget to specify dependency information.

Third, state of the art solutions, e.g., HomeOS [7] can detect conflicts when two apps try to access the same device at the same time. However, in a home many devices available for the control and it is possible for two apps to use two separate devices and still conflict with each other, e.g., one app is running a humidifier while another app is running a dehumidifier in the same room. Therefore, an important challenge is to determine how to detect control dependency of the actuators that arise from running different devices.

Fourth, based on large scale and long term deployment experience in residential homes [1], we know that sensors just don't die, they experience failure in a variety of ways. It is important to detect not only *fail-stop* failures, but *obstructed-view* and *moved-location* failures; new types of failures common in smart homes that are very difficult to detect even if sensors are made highly reliable and barely addressed to date in the literature or in real deployments. Since all the apps will share a single set of sensors, we can not use redundant sensors to assess sensor failure status. We also need to minimize training effort of the users. Therefore, it is an major challenge to determine how to detect these realistic sensor failures in a smart home with minimal training effort and without requiring sensor redundancy.

Fifth, humans are intimately involved with the integrated smart home system. Apps that perform interventions can control the physiological parameters of the human-in-the-loop in a conflicting way, e.g., one app administers a drug that decreases the heart rate while another app suggests to do exercise which increases the heart rate. It is very challenging

to detect such human-in-the-loop dependencies as we can not ask the app developers to specify all the physiological parameters that can be affected by their interventions. Because they would have to specify hundreds of physiological parameters for each intervention as human body has many interconnected parts. The major challenge is to detect such human-in-the-loop dependencies with minimal efforts of the app developers. Another challenge is to make sure that the dependency analysis can be personalized, e.g., if the user has kidney related problems, the user, or the caregiver, or the doctor should be able to configure the dependency analysis to focus on the kidney.

Sixth, there may be apps with real-time constraints for reliable packet delivery within latency bounds. It is hard to satisfy such a constraint since wireless links are highly non-deterministic because of link burstiness and interference. Link burstiness is a physical property which means that transmissions on a wireless link do not have independent probability of failure; instead they fail in a burst. Link interference is another physical property of the communication environment which causes packet transmission between different links to interfere with each other which results in packet loss. Therefore, it is a major challenge to determine how to compute latency bounds for transmitting real-time periodic streams over wireless networks by addressing dependencies on the link burstiness and interference.

1.3 Thesis Statement and Solution Overview

1.3.1 Thesis Statement

Our **hypothesis** is that *by applying new and expressive app metadata, combined with sensor-appliance behavioral patterns and link characterization can significantly enhance systems of systems dependency analysis including dependencies at the actuator level, involving humans-in-the-loop, sensor reliability status, and for real-time communication over wireless networks as compared to today's interoperability techniques and systems.*

1.3.2 Solution Overview

An overview of the solution is shown in Figure 1.1. App developers specify dependency information as meta data within their apps and their apps are uploaded to an app store. Users can choose and install apps from the app store. 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, A3) deployed in the home can be plugged into the system and the Device Plugins layer contains the device drivers. We consider humans as sensors as they can specify their preferences and change app parameters. DepSys has many components to address many dependencies and the components are described in detailed in Chapter 3.2. However, among many components, four components have major functionalities to prove the hypothesis. The four major components are: Actuator Control Dependency Checker, Sensor Failure Detector, Human-in-the-loop Manager,

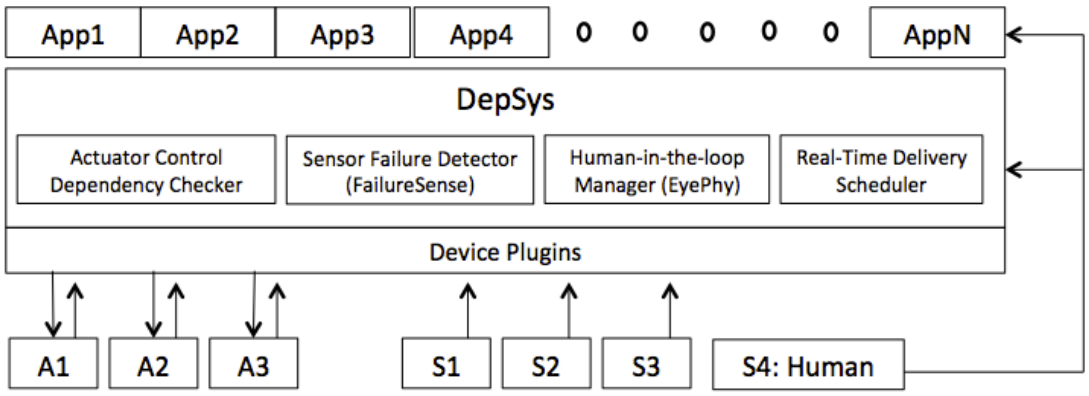


Figure 1.1: Conceptual Integration of four components: Actuator Control Dependency Checker, Sensor Failure Detector, Human-in-the-loop Manager, and Real-Time Delivery Scheduler in the DepSys platfrom.

and Real-Time Delivery Scheduler. Figure 1.1 shows a conceptual integration of these components in the DepSys platform.

When an app wants to control an actuator, the control request has to go through Actuator Control Dependency Checker, which detects conflicts within and across actuators. Sensor Failure Detector keeps track of the sensors that are being used by each app and when a sensor experiences *fail-stop*, *obstructed-view*, or *moved-location* failure, it reports the failure to the corresponding app. It uses FailureSense solution (Chapter 5) to detect sensor failures. Human-in-the-loop Manager detects conflicting and interacting interventions of the apps by considering a wide range of physiological parameters of the human-in-the-loop (i.e., the user) using EyePhy (Chapter 6). Finally, the Real-Time Delivery Scheduler addresses apps' dependencies on real-time constraints by scheduling packet transmission of the sensors and actuators for reliable packet delivery within latency bounds. The latency bounds can be considered a requirement from an app and Real-Time Delivery Scheduler tries to satisfy such a requirement. The way it characterizes wireless links and schedules packet transmission is described in Chapter 7.

1.4 Contributions

The contributions of this dissertation are the following:

1. DepSys provides the most comprehensive strategies to specify, detect, and resolve conflicts in a home setting to date by addressing a spectrum of dependencies including requirements, name, control, sensor reliability, and human-in-the-loop dependencies. In addition, DepSys handles the case when app developers fail to specify dependencies.
2. DepSys automatically resolves control conflicts of sensors and offers a strategy for resolving control conflicts of actuators that reduces cognitive burden of the users and allows apps to be run in a more flexible way than the state

of the art solution. We are the first to demonstrate that dependency metadata that focuses on the effect on the environment enables us to detect conflicts across devices that isn't possible by monitoring actuations of individual devices as performed by the state of the art solutions. Also, our proposed novel metadata helps understanding context and resolving actuator level conflicts more accurately that couldn't be achieved through existing priority based solutions.

3. We are the first to show that sensor-appliance behavioral patterns could be exploited to detect not only *fail-stop* failure, but *obstructed-view*, and *moved-location* failures that are realistic sensor failures in smart homes, but barely addressed to date in the literature or in real deployments. Our failure detection solution, FailureSense overcomes several limitations of the state of the art solutions, including it requires much less training effort, it is scalable in detecting multiple sensor failure even if they fail simultaneously, and it doesn't require sensor redundancy.
4. Our human-in-the-loop dependency checker, EyePhy provides the most comprehensive dependency analysis across human-in-the-loop apps' interventions to date by modeling the environment, i.e., the human body using over 7800 variables and using the model as a simulator to understand the potential interactions of interventions. Our solution takes into account drug dosage and time gaps between the interventions at the dependency analysis, it reduces app developers efforts in specifying dependency metadata, and it offers personalized dependency analysis for the user.
5. We perform a 21 days long empirical study where we transmit 3,600,000 packets over every link of a 802.15.4 testbed to understand the burst behavior of wireless links. Based on the findings, we define a new metric *max burst length (Bmax)* to characterize link burstiness and design a static network-wide stream scheduling algorithm that uses a novel *least-burst-route* to produce latency bounds of real-time periodic streams by taking into account link bursts and interference. Empirical evidence shows that stationarity of link quality can be better characterized by $Bmax$ than state of the art metric PRR.
6. 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. We also evaluate DepSys's performance in addressing other dependencies, including sensor failure detection and ability to bound the latency of packet delivery using real world data that demonstrates its superiority over the state of the art solutions.

1.5 Caveats

Our solution comes with a few caveats:

- 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.

- There may be conflicts due to app interdependencies 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 thesis.

- Sensor failure detection algorithm may not detect a *moved-location* failure if the sensor is slightly moved from its location. Note that small sensor displacement may not make any difference in application semantics. Hence, this type of minor movement is not a movement failure and is not detected as such.

- The performance of the sensor failure detection algorithm may degrade if the behavior of the residents change after the training period. Putting humans into the loop will allow the system to use human feedback to distinguish between behavior change and sensor failure over time. We consider it future work.

- The latency bounds generated by Real-Time Delivery Scheduler depends on the topology. If the wireless links are not good enough to satisfy our wireless communication model after network characterization, then some nodes may need to be moved or new nodes may need to be added to create the "right" topology having good burst properties to satisfy the model assumption.

- The evaluation is limited in terms of number of sensors, people, and houses involved. For example, sensor failure detection evaluation uses only 71 days of data collected from 3 real homes. Also, we need an app store for the home to evaluate DepSys. There is no such well-established app store for the home to date. Hence, we create 35 apps from various categories, including energy, health, security, and entertainment that will serve as our app store. We use these 35 apps and 219 days of data collected from WSU CASAS smart home project [10] to evaluate the effectiveness of actuator control dependency detection and resolution. Hence, the evaluation results only show a proof-of-concept of the DepSys design. Further investigation and deployment are required to understand its usability.

1.6 Organization of the Dissertation

The rest of the dissertation is organized as follows:

- Chapter 2 discusses state of the art solutions related to DepSys and their limitations.

- Chapter 3 provides an overview of the DepSys system design and its implementation.
- Chapter 4 describes evaluation of DepSys in terms of its conflict detection and resolution capability at the actuator level.
- Chapter 5 describes how DepSys addresses dependencies on sensor reliability status by using a novel sensor failure detection algorithm named FailureSense and its evaluation.
- Chapter 6 describes human-in-the-loop dependency addressing mechanism named EyePhy and its evaluation.
- Apps may have dependencies on real-time constraints for delivering packets over a wireless network reliably and within latency bounds. Chapter 7 describes how DepSys addresses such dependencies and its evaluation.
- Chapter 8 concludes the dissertation by summarizing the contributions and discussing possible future work.

Chapter 2

Related Work

This chapter presents the state-of-the-art solutions related to the challenges that DepSys addresses. Although DepSys addresses many dependencies, its major challenges are to address dependencies at the actuator level control, on sensor reliability status, involving human-in-the-loop, and on real-time constraints for delivering packets within latency bounds reliably over wireless networks. All of these depend on the architecture of a smart home. In this chapter, first, we describe several smart home systems and proposed smart home architectures (Section 2.1). Second, we describe commercial home automation systems that are available in the market and their limitations (Section 2.2). Third, we compare DepSys with the component based integration techniques (Section 2.3). Fourth, we describe several mechanisms that are proposed in the literature for addressing control dependencies at the sensors and actuators (Section 2.4). Finally, we describe the related work of several other challenges that DepSys addresses including sensor reliability dependencies (Section 2.5), human-in-the-loop dependencies (Section 2.6), and dependencies on real-time constraints for delivering packets within latency bounds over wireless networks (Section 2.7).

2.1 Smart Home Systems Research and Architectures

Several smart home systems have been proposed and developed over the years. However, the variations among the proposed smart home architectures are limited. Georgia Tech's Aware Home [11] is a prototype home laboratory that provides context awareness and ubiquitous sensing. Their Smart Floor project [12] uses footsteps to determine *who* is *where* to support ubiquitous computing. However, their solution relies on the installed infrastructure, e.g., they require smart floor tiles to be installed that can measure force for sensing footsteps. Also, in their paper, they do not explicitly address smart home architectural issues, e.g., whether the sensors will form a single hop or a multihop network, how to add new sensors, actuators, and applications, and whether multiple applications can share sensors and actuators.

MIT's PlaceLab [13] is a living laboratory to study ubiquitous technologies in a home setting. Their 1000 square foot lab contains a wide varieties of sensors. This is a great tool for researchers to collect data for understanding and developing context-aware and ubiquitous interaction technologies. However, it is not targeted for end users. As a result, the paper does not address several smart home architectural issues, e.g., how to add new applications, how the applications share sensors and actuators, and how the users can specify their preferences.

Harvard's CodeBlue [14] is a wireless infrastructure for emergency medical care. It is designed to scale thousands of devices and to work in extremely volatile network conditions in a hospital setting to assess patients by using low power, wireless vital sensors, PDAs, and PCs. The architecture provides support for reliable and ad hoc routing, flexible node naming and discovery, authentication and encryption. Although it is designed to work a hospital setting instead of a home, it helps shaping subsequent home health care systems.

University of Virginia's *AlarmNet* [8] combines the ubiquitous sensing capabilities of PlaceLab with the medical support of CodeBlue, but is tailored for long term monitoring of health conditions of the elderly for assisted-living in residential buildings. It learns the activity patterns of the residents and uses it to perform context aware power management to prolong the battery life of the sensors and to adjust privacy policies dynamically. Its architecture integrates heterogeneous custom and commodity sensors in a scalable way. The deployed sensors transmit data over a single or multi-hop wireless network to a gateway. The gateway program is run on an embedded platform, e.g., Crossbow stargate, which provides two way communications between IP and wireless sensor networks. The backend program collects all the sensor data streams through the gateway, processes them, and produces new data streams that are saved in the back-end database for subsequent use by the other modules. However, it has been difficult to add heterogeneous sensors to the system with this architecture. To overcome this limitation, a second generation architecture, Empath [9] is proposed, where each (sub)system produces data for a web server and integrated system accesses these data through the web server. By monitoring resident's sleep quality, weight, activities of daily living, and speech prosody, Empath provides caregivers more accurate and thorough information about several medical conditions of the resident, including depression, epilepsy, and incontinence.

University of Florida's Gator Tech Smart House [15] offers service-oriented programmable pervasive spaces. Their architecture provides a middleware that contains separate sensor/actuator physical layer, sensor platform layer, service layer, context management layer, and knowledge layer. At the top layer of the architecture, there is an application layer that supports activation and deactivation of services. Developers can browse and discover services, add new services, and use them to build remote monitoring and intervention services for any pervasive computing environment.

Microsoft's HomeOS [7] provides a PC-like abstraction for the technologies in the home. It treats network devices in the home, e.g., locks, lights, thermostats, sensors as peripherals connected to a single logical PC. HomeOS proposes an app based paradigm, where app developers develop smart home apps that are uploaded to an app store named HomeStore. Users can download apps from the HomeStore and install them in their PCs using HomeOS. HomeOS uses

a layered architecture. At the lowest level, it has Device Connectivity Layer (DCL), which contains protocol specific device drivers. In the second layer, it has Device Functionality Layer (DFL), which is protocol independent. It provides high level abstractions that are independent of low-level details of the devices to the app developers, which eases their application development. In the third layer, it has Management Layer, which provides a management interface to the users, where they can control *which* app can access *which* devices at *which* time. The top layer is the Application Layer, where HomeOS runs apps. HomeOS kernel is completely agnostic to the installed devices and apps, which allows incremental addition of devices and apps. DepSys uses an app based architecture for the integrated system of the home by being motivated from HomeOS and smart phones [16] [17]. Although the DepSys architecture is similar to that of HomeOS, it has additional features for comprehensive dependency detection and resolution across apps.

2.2 Commercial Home Automation Systems

Many commercial home automation systems are available on the market that integrate multiple devices in the home, e.g., ADT [18], Vivint [19], Honeywell [20], Bosch [21], Philips [22], SimpliSafe [23], Viper [24], Alarm.com [25], Monitronics [26], M1 Security & Automation Controls [5], Control4 [3], HomeSeer [4], and Leviton [6]. However, they have a few shortcomings. First, most of these systems are tailored for individual purposes. For example, there are systems targeted for security [23], for energy management and security [18] [19] [25] [26], and for energy management, security, and home health care [20] [22]. DepSys provides a generic platform for running all types of applications in a smart home. Second, most of these commercial home automation systems either do not allow adding, removing functionalities after they are installed, or even if they do, it is very difficult for the end users to do so. The users have to purchase the complete package in advance before even knowing if the solution will fit their lifestyles. DepSys allows users to add new devices and apps incrementally based on their needs in a more flexible way. Third, most of these commercial home automation systems systems are monolithic, they come with a fixed set of apps designed by the same vendor not to conflict with each other, and are not extensible to external app developers [18] [6] [25] [24] [20]. DepSys allows the running of apps developed by external app developers. But it requires app developers to specify dependency metadata for performing comprehensive dependency analysis across apps. However, there are some platforms, e.g., HomeOS [7], Control4 [3], and HomeSeer [4] that are extensible to external app developers. But they lack mechanisms to perform comprehensive dependency analysis across apps that DepSys offers. For example, HomeOS uses app priority to resolve control conflicts at the actuator level. 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. DepSys requires the app developers to specify additional metadata called *emphasis*, which allows DepSys to detect and resolve such conflicts more accurately. Also, existing state of the art solutions are limited in detecting conflict that occurs within a device, e.g., two apps are controlling the same light at the same time. 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 metadata called *effect*, *emphasis*, and *condition*. Also, DepSys provides mechanisms to address dependencies on sensor reliability status, involving human-in-the-loop, and on real-time constraints that commercial home automation systems do not offer.

2.3 Component based Integration Techniques

There are tools that allow developing systems by integrating components. For example, Ptolemy [27] is a framework for integrating various components into a single system. Each component needs to be described using formal description with sufficient details so that Ptolemy can generate code automatically. Ptolemy also provides a run-time infrastructure to mediate between components. However, it is a very generic tool and it does not provide any insight about how to deal with various types of dependencies that may arise among multiple simultaneously running applications in a home setting, e.g., how to detect conflicts that arise across devices, how to consider device semantics to resolve conflicts, what will be the role of the user etc. Also, it can not deal with the case if the app developers forget to specify dependencies that DepSys can. There are other tools [28] [29] that allow tracing intra and inter-component event and data dependencies or try to provide correctness by construction to meet certain properties, e.g., global deadlock-freedom, individual deadlock freedom of components and interaction safety between components [30]. But these are also general tools that do not provide any additional insights for addressing various dependencies among applications in a home setting. Also, such component based analysis techniques usually assume that components are known during offline static analysis and are not flexible enough to include new components or apps at runtime dynamically.

2.4 Control Dependencies involving Sensors and Actuators

There has been several works on running concurrent applications in sensor networks [31] [32] [33] [34] [35]. But most of these works do not address conflicting control requests from multiple applications regarding the use of sensors and actuators. For example, TinyCubus [31] offers a framework for TinyOS-based [36] sensor networks. Its infrastructure addresses challenges involving network device heterogeneities and evolution of applications. It enables components to be installed dynamically and supports data sharing among components to achieve cross-layer optimizations. When an application requires new processing or aggregation function, TinyCubus distributes and installs code in the network.

To do that, it assigns roles to each node. A role defines the function of a node based on its properties, e.g., hardware capabilities and location. The roles are used to address device heterogeneity issues, as the nodes that actually need a component will receive and install it. TinyCubus partition the network into different groups based on the roles of the nodes, where each group executes one application. However, it does not support node level concurrent applications. Hence, TinyCubus do not address the case if multiple applications require the usage of sensors in a conflicting way. Melete [34] enables execution of concurrent applications at both node and network levels. At node level, it runs Maté virtual machine [37] that allows multiple applications to reside on a sensor node and concurrent execution of these applications. At network level, it uses Trickle [38] for efficient code dissemination across a network, where multiple applications can be deployed at some portions of the network. But, Melete ignores the problem of control conflicts for using the sensors.

However, there are some works that deal with control conflicts at the sensors and actuators. For example, [39] defines a policy language for sensor network management using a language called APPEL [40]. They identify three types of conflicts in a sensor network: general policy conflicts (e.g., one policy wants an accelerometer data at 100 Hz while another policy wants the same accelerometer data at 200 Hz), goal conflicts (e.g., maximize sensor battery life vs. report all the anomalies promptly), and external resource conflicts (e.g., two applications competing for bandwidth or processing power). APPEL allows a policy to have a preference (e.g., priority). Allowed preferences are 'must', 'should', 'prefer', and 'don't care'. At the time of a conflict, the most preferred policy is chosen for the conflict resolution. PhysicalNet [41] provides a generic paradigm for managing and programming world-wide distributed heterogeneous sensor and actuator resources in a multi-user and multi-network environment. By allowing owners to specify fine grained use-based access rights control and conflict resolution mechanism, PhysicalNet allows sharing of resources and increases number of concurrent applications running on the devices. When multiple users simultaneously specify contradictory requirements on the same resource, PhysicalNet 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. Access rights and resolvers can be dynamically changed according to the behavior of other services. Bundle [42] proposes a group based programming abstraction for cyber-physical systems. Bundle eases programming effort, supports heterogeneous sensors and actuators, allows intra and inter network mobility, and enables multiple applications to use the same sensors and actuators. It uses conflict resolution policy as in PhysicalNet [41]. HomeOS [7] enables multiple apps to run in a smart home to control the sensors and actuators. When a user installs a new app, HomeOS asks the user to specify access control rules for the app. HomeOS uses Datalog [43] rule database to detect if the new application could access a device at the same time as other app. If that happens, it asks the user to specify which app should have a higher priority. At runtime, it resolves conflicts in favor of the higher priority app. However, using priority is not enough for conflict resolution at the actuator level as shown with an example in Section 2.2. Comparing with these solutions, DepSys offers more fine grained control

dependency detection and resolution at the actuator level that these solutions cannot provide, e.g., control dependency detection across appliances, differentiating false conflicts from true conflicts by using additional metadata, and resolving accurately. Also, DepSys addresses control dependencies of the sensors by considering three dimensions: app priority, sensor resource availability constraint, and app requirements.

2.5 Sensor Reliability Assessment

Several techniques have been proposed to detect *fail-stop* failure of sensors. In Memento [44], nodes in the network cooperatively monitor each other to implement a distributed *fail-stop* failure detector. Each node runs the Memento protocol, which uses existing routing topologies and other protocol's beacons as heart-beat messages to detect a sensor failure. Memento suggests each node to report only the status change instead of sending status messages at a fixed period, which saves communication bandwidth and energy consumption of the sensor nodes. LiveNet [45] uses passive sniffers co-deployed in the network to reconstruct dynamics of sensor network deployments. Sniffers record traces of all packets that are received on the radio channel. LiveNet merges these traces into a single one to obtain a global picture of the network's behavior, including node failures. One advantage of using LiveNet is that its infrastructure can be deployed, reconfigured, and torn down independently from the network under test. Because, it doesn't require any change of the codes of the sensor nodes under investigation. MANNA [46] offers a policy-based network management system. It assigns different roles (network managers or agents) to various sensor nodes. Based on the roles, these nodes request or response messages with each other for various management purposes, including fault management. Sympathy [47] is a tool for detecting and debugging failures in sensor networks. In Sympathy, each live node generates two types of data: it's own periodic traffic and Sympathy generated traffic. Code running at the sink detects a failure when a node generates less own traffic than expected. However, it requires changes in the codes of the sensor nodes to transmit Sympathy protocol packets periodically. Also, it is not designed for event-driven applications. These techniques are mainly suitable for low-level debugging and detecting *fail-stop* failure, but not designed to detect *non-fail-stop* failure, e.g., *obstructed-view* and *moved-location* failures in an event-driven system.

Several techniques have also been proposed to address *non-fail-stop* failure. A sensor may experience *non-fail-stop* failure in various ways and a taxonomy of common sensor data faults is defined in [48]. SMART [49] detects *non-fail-stop* failure by analyzing the relative behavior of multiple classifier instances trained to recognize the same set of activities based on different subset of sensors. SMART requires non-trivial effort in training and it is not scalable in detecting multiple sensor failure if they fail within a short time frame. Reputation-based Framework for Sensor Networks RFSN [50] detects *non-fail-stop* failure by exploiting the correlation between neighboring sensors. But it requires sensor redundancy to build meaningful correlation and it doesn't work if the neighboring sensors are compromised or failed. Also, as SMART [49] indicates, sensors are often correlated in a different way based on

the activity performed and looking at just temporal correlation is not a good approach to detect *non-fail-stop* failure. FIND [51] detects *non-fail-stop* failure under the assumption that sensor readings reflect the relative distance from the nodes to the event. It works where the measured signals attenuates with distance, e.g., a system that captures acoustic signals. It also requires redundant sensors to compare relative distances. Compared to these solutions, our solution detects *obstructed-view*, *moved-location*, and *fail-stop* failures without redundancy, with much less training effort, and it is scalable in detecting multiple sensor failure even if they fail simultaneously.

Detection and classification of electrical appliance activation events is not our main focus. Several state of the art techniques are available for this purpose that use single-point sensing [52] [53] [54], distributed sensing [55], or a combination of these two [56]. We prefer to use single-point sensing for our work, because it doesn't rely on the deployed sensors in the home for which we are trying to assess failure.

2.6 Human-in-the-Loop

Human-in-the-loop is an active area of research. A taxonomy of human-in-the-loop applications and challenges for incorporating human models into formal methodology of feedback control are specified in [57]. Control designs with a human as part of the loop have been used in physiological control systems [58, 59], mobile sensing and computing systems [60–63], thermal control systems [64–66], and robotic systems [67]. Although these works address different human-in-the-loop issues, these issues are different from the human-in-the-loop dependencies that DepSys addresses. We mainly deal with the case when applications perform interventions on a human body and the dependency issues raised by such interventions.

Several works have been done involving theory, characterization, and selection of interventions to change human behavior. [68] studies how to apply health behavioral theories into mobile interventions. Based on the current literature on mobile technology health behavior interventions, the authors identify four major areas of health behavior where interventions are performed. These areas are smoking, weight loss (including diet and exercise interventions), treatment adherence, and chronic disease management. The authors determine that current theories are inadequate to inform mobile intervention development. For example, consider that an app suggests to do exercise. The authors mention that there has been a few studies to determine how long the app needs to wait before prompting again and how to update the timing and intervention message based on prior responses. As mobile interventions are becoming adaptive and interactive, intervention theories need to be advanced to answer many intervention development questions. The authors develop a human-in-the-loop feedback control model to intervene smoking urges to demonstrate the need for dynamic feedback control system theories. A technique for characterizing and designing interventions is described in [69]. A taxonomy of the behavior change techniques used in interventions is specified in [70]. [71] suggests to use theories in designing interventions to change human behavior. The paper lists a set of behavior change techniques and

develops a procedure to identify relationships between the behavior change techniques and behavioral determinants. [72] provides a technique to assess the extent to which a behavioral intervention is theory based. These works may help app developers to choose appropriate interventions to change the user's behavior. However, DepSys does not use any of these techniques to address app dependencies due to conflicting interventions. Instead, it uses a human physiological model named HumMod [73] as described below.

There are many physiological models enabling the simulation of different functionalities of a human body including the cardiovascular system [74], pulmonary system [75], blood circulation [76], cardiac metabolism [77], and microcirculatory hemodynamics in vascular networks [78]. However, these models are developed to understand the behavior of individual organs without providing the integration across different organ systems and simulation of the whole body. There are a few physiological simulators that integrate different organs. For example, Nottingham Physiology Simulator [79] allows simulation of cardiovascular, acid-base, respiratory, and renal physiological models. Guyton [80] and HUMAN [81] are two historical models of integrative physiology. QCP [82] is an extension of HUMAN and it incorporates cardiovascular, renal, respiratory, endocrine, and nervous systems. QCP is developed in C++, which limits the ability to update the physiological models, e.g., changing or adding an equation requires recompilation of the whole simulator. HumMod [73] is developed to overcome this limitation as it uses XML files to describe the model parameters and the quantitative relationships among them. It uses over 7800 variables to capture cardiovascular, respiratory, renal, neural, endocrine, skeletal muscle, and metabolic physiology. The model it uses is developed using the empirical data collected from peer-reviewed physiological literature. HumMod provides a model to understand the complex interactions of integrative human physiology and allows the simulation of different interventions on the human body.

2.7 Real-Time Wireless Communication

Several research works have already been done to characterize wireless links. Statistical properties of low power wireless links have been analyzed in [83]. It shows that link failures have temporal characteristics. Characterization of signal strength properties is demostrated in [84] by using Monopole Antennas. It shows that antenna orientation effects are the dominant factor of the signal strength sensitivity. [85] presents a metric called competence that characterizes link for a longer time frame. It improves end-to-end delivery ratio and reduces energy consumption. E^2WFQ [86] offers fair packet scheduling. It is an energy saving version of Weighted Fair Queueing (WFQ) algorithm. ATPC [87] employs feedback-based dynamic transmission power control to adjust the quality of radio communicaiton. It also achieves more energy saving. ART [88] controls topology to reduce power consumption and channel contention. It can adapt to the variations in link quality and contention. [89] exploits spatial diversity in industrial wireless networks based on relaying approach. [90] presents a model that accounts for the correlation that exists in shadow fading between links in multihop networks. [91] presents Radio Irregularity Model (RIM) to analyze the impact of radio irregularity on

the communication performance and provides solution to deal with this issue. [92] offers algorithm admission control and two scheduling policies that are proved to be feasibility optimal for wireless network based applications with QoS requiements.

There has been studies that shows that most of the wireless links are bursty. Srinivasan et al. [93] have studied 802.15.4 and 802.11b networks and presented a new metric β to measure link burstiness. He has shown that we can avoid link burstiness by having an interpacket delay of 500 ms. But our 21 days of study on a 802.15.4 network shows that we can do even better for some links i.e. for some links we don't have to wait that much of time to avoid packet loss due to link bursts.

WirelessHART [94] is a wireless mesh networks communication protocol designed to meet the needs for process automation applications. It combines several features to provide 99.9% end-to-end reliability in all industrial environments. The features include channel hopping on a message-by-message basis, monitoring paths for degradation and automatic repair, finding alternative paths around obstructions. But it does not address packet loss due to link bursts. As a result a latency bound cannot be provided.

There are a number of scheduling algorithms for stream transmission. But they have many assumptions or shortcomings that make them unsuitable for scheduling periodic streams with real-time constraints in WSN. Previous performance analysis [95] shows that DSR [96] outperforms other ID-based routing protocols in terms of successful packet delivery ratio, but it does not consider time constraints. RAP [97] uses a velocity monotonic scheduling algorithm that takes into account both time and distance constraints. It maximizes number of packets meeting their end-to-end deadlines, but reliability aspects of individual streams are not addressed. SPEED [98] maintains a desired delivery speed across the sensor network by a combination of feedback control and non-deterministic geographic forwarding. It is designed for soft real time applications and it is not concerned with the reliability issues of individual streams. Also, our approach is based on static scheduling, while these approaches are based on dynamic scheduling. So, neither of these approaches seem to be appropriate to be compared against our approach.

RI-EDF [99] is a MAC layer protocol that provides a real-time guarantee by utilizing the rules of EDF [100] to derive a network schedule. But, in this work the network has to be fully linked, i.e., every node has to be within direct transmission range of every other node. This constraint makes this protocol unsuitable for many networks. [101] provides timeliness guarantees to multi-hop streams by explicitly avoiding collisions and scheduling messages based on per-hop timeliness constraints in real-time robotic sensor applications. None of the above mentioned algorithms actually considers link burstiness that can affect the packet transmission significantly.

2.8 Summary

This chapter presents the state of the art techniques related to the challenges addressed by DepSys. We cover state of the art smart home systems in the literature, commercial home automation systems and their limitations, techniques to address control dependencies at the sensor and actuator level, sensor failure detection techniques, related human-in-the-loop systems, and techniques to characterize wireless links for real-time packet delivery.

Chapter 3

DepSys Platform Design and Implementation

The design of DepSys is based on two fundamental observations. First, integrating Cyber-Physical Systems (CPSs) of smart homes will offer innumerable advantages. Second, app based paradigm in smart phones and tablets is getting very popular now-a-days. We describe the basis of these observations and how these two apparently irrelative observations tailor the design of DepSys below.

As sensors and actuators network mature and Cyber-Physical Systems (CPSs) enable systems of systems to control physical world entities, it opens a new horizon for home automation. There is an ever increasing number of systems installed in smart homes under various application domains including home health care, smart energy management, security, and entertainment. Whereas some systems may share resources, usually each application domain is tailored for its individual purpose and treated as an independent system. For example, sleep monitoring and depression monitoring systems may both share the sensors deployed in the bed to monitor quality of sleep, but usually these two home health care systems are independent of any energy management system that tries to save energy. As mentioned in Chapter 1, if we can integrate all the Cyber-Physical Systems in a smart home under a common platform, then there will be many advantages. For example, if we can integrate the energy management system with home health care system, then the energy management system can adjust room temperature depending on the physiological status of the residents as detected by the home health care system. Also, integration will allow avoiding 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 will be able to share sensors, which will reduce cost of deployment, improve aesthetics of the rooms, and reduce channel contention for packet transmission.

As smartphones and tablets are becoming popular, app based paradigm has changed the way we interact with smart

devices. Google announced that a total of 50 billion apps had been downloaded from the Google Play, formerly known as Android Market, by the end of July 2013 and currently (July 2014) 1.3 million apps are available in the Android Market [102]. Similarly, more than 75 billion apps have been downloaded from Apple's App Store by June 2014 and 1.2 million apps are available there for the download [103]. An estimated 1.2 billion people around the world were using mobile apps at the end of 2012 [104]. Being motivated by the huge success of 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 there are commercial home automation system available in the market, e.g., ADT [18], Leviton [6] that come with apps, these apps are built by the same vendor designed not to conflict with each other. Their platform is not open to the external app developers. However, there are some platforms that are open for the external app developers, e.g., HomeOS [7] and Control4 [3]. But these state of the art solutions lack capability in detecting and resolving conflicts that DepSys offers. Some concrete limitations of the state of the art solutions are described in Chapter 2.2.

3.1 Contributions

The contributions of DepSys are the following:

- DepSys provides the most comprehensive strategies to specify, detect, and resolve conflicts in a home setting to date by addressing a spectrum of dependencies including requirements, name, control, sensor reliability, and human-in-the-loop dependencies. In addition, DepSys handles the case when app developers fail to specify dependencies.
- DepSys automatically resolves control conflicts on sensors by considering three dimensions: app priority, resource availability, and app requirement.
- DepSys is the first solution to demonstrate that dependency metadata about the effect on the environment enables us to detect conflicts across actuators that isn't possible by monitoring actuations of individual devices as performed by the state of the art solutions. Also, our proposed novel metadata helps understanding context and resolving actuator level conflicts more accurately that couldn't be achieved through existing priority based solutions.
- DepSys reduces cognitive burden of the users to specify the conflict resolution policy of actuators, which also allows apps to be run in a more flexible way than state of the art solutions.

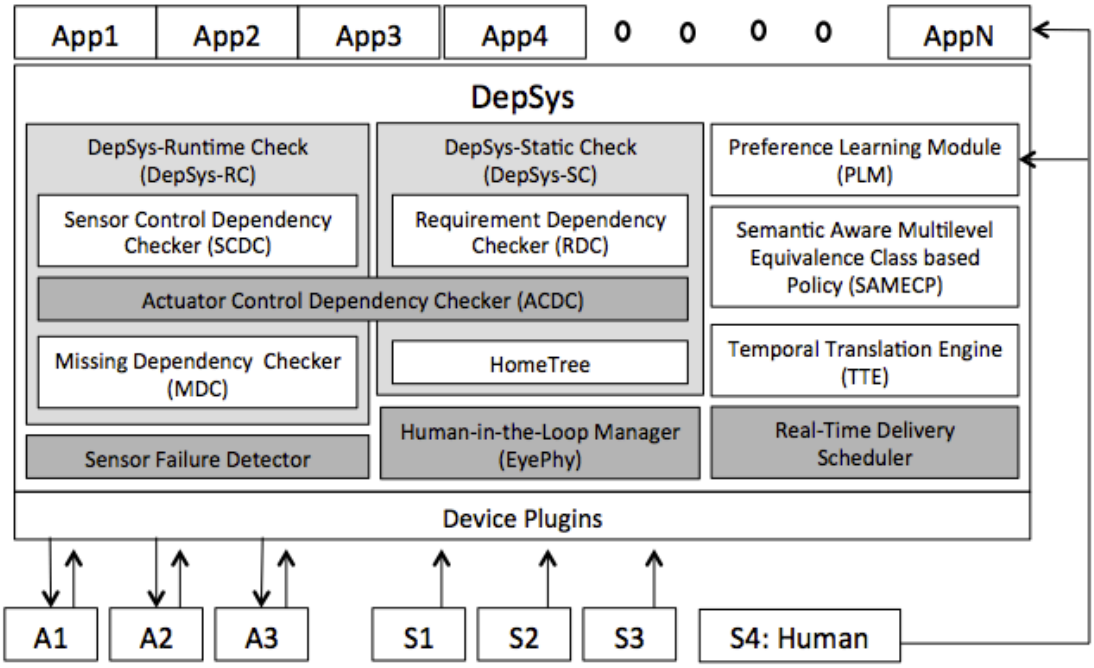


Figure 3.1: DepSys System Architecture.

- By using 219 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.

3.2 System Architecture

The DepSys system architecture is shown in Figure 3.1. 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, A3) deployed in the home can be plugged into the system and the Device Plugins layer contains the device drivers. 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. When a new app is installed, DepSys uses Actuator Control Dependency Checker (ACDC) to check for actuator level dependencies across previously installed apps. If a potential conflict is detected, the Preference Learning Module (PLM) learns user preferences and allows users to choose a policy for conflict resolution. DepSys-Runtime Check (DepSys-RC) contains three modules: Actuator Control Dependency Checker (ACDC), Sensor Control Dependency

Checker (SCDC), and Missing Dependency Checker (MDC). ACDC and SCDC detect and resolve control dependencies of the actuators and the sensors, respectively, using user preferences and policies specified in the PLM. Note that ACDC module spans both in DepSys-SC and DepSys-RC as control dependencies at the actuator level are checked both at app installation time and at runtime. The MDC addresses the missing dependency problem, i.e., when the app developer does not specify a dependency that actually exists. Sensor Failure Detector keeps track of the sensors that are being used by each app and when a sensor experiences *fail-stop*, *obstructed-view*, or *moved-location* failure, it reports the failure to the corresponding app. To detect a sensor failure, it uses FailureSense solution (Chapter 5). Human-in-the-loop Manager detects conflicting interventions among apps due to human-in-the-loop using EyePhy (Chapter 6). Finally, the Real-Time Delivery Scheduler schedules packet transmission of the sensors and actuators by considering link burstiness and interference so that the packets are delivered to the apps within the latency bounds. The latency bounds can be considered a real-time constraint from an app and Real-Time Delivery Scheduler tries to satisfy such a requirement. The way it characterizes wireless links and schedules packet transmission is described in Chapter 7.

3.3 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:

   (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. **Sensor Reliability Dependency:** It arises when apps rely on sensor reliability status. For example, *App1* uses motion sensor *M1* and if *M1* fails, the performance of *App1* will degrade.
7. **Human-in-the-Loop Dependency:** It arises when multiple apps perform interventions that change one or more physiological parameters of the involved human-in-the-loop (i.e., the user) in a conflicting way, e.g., one app administers a drug that decreases the heart rate while another app suggests to do exercise which increases the heart rate.
8. **Dependency on Real-Time Constraints:** It arises when apps have real-time constraints. There can be many types of real-time constraints. In this thesis, we only address the case where apps require reliable and on time packet delivery from the source nodes to the destination nodes over a wireless network. The solution needs to address dependencies on two physical physical properties of the communication environment, link burstiness and interference.
9. **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 dependencies in this thesis.

In this chapter, we specify how we address the dependencies from $\#1$ to $\#5$. Solutions to address dependencies $\#6$, $\#7$, and $\#8$ are designed and evaluated separately and are described in chapters 5, 6, and 7, respectively.

3.4 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 [9], 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.

3.5 Addressing Name Dependency

We have two separate design choices to address name dependencies (c.f. Section 3.3). 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 3.9.3).

3.6 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 [16] allows apps to choose a rate from a set of only 4

available rates. PhysicalNet [41] 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 3.9.2). *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 Checker (SCDC) 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: $selected\_rate \leftarrow r\_max$
2: $S \leftarrow sort(rates)$
3: **for** $i \leftarrow 1$ to n **do**
4: $g \leftarrow GCD(S(i), S(i+1), ..., S(n))$
5: **if** $g < selected\_rate$ **then**
6: $selected\_rate \leftarrow g$
7: **break**
8: **end if**
9: **end for**
10: **for** $i \leftarrow 1$ to n **do**
11: $rr_i \leftarrow floor(S(i)/selected\_rate) * selected\_rate$
12: **end for**
13: $resolved\_rates \leftarrow sort([rr_1, rr_2, rr_3, ..., rr_n])$
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*.

3.7 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 [7] 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.

3.7.1 *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 in the same room. 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 [105], 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.

3.7.2 *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 Chapter 2.2 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 3.9.3.

App developers specify *emphasis* in the XML metadata 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.

3.7.3 *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.

3.7.4 Limitations of *Effect*, *Emphasis*, and *Condition*

Our proposed metadata, *effect*, *emphasis*, and *condition* are powerful enough to detect conflicting actuations that occur within and across devices. However, they have a few limitations.

In order to use *effect* metadata, the app developers need to know the effects of their control actions. Also, the number of possible effects need to be limited so that it incurs minimal effort for them to specify all the *effects* of their actuations. There are CPS areas where none of the two assumptions hold, e.g., when apps perform interventions that affect a human body. Because, in this case, all the *effects* of an arbitrary intervention are not known by the medical research yet and hundreds of parameters can be affected by each intervention. Even if the possible effects are limited, sometimes specifying *effects* can be confusing. For example, an app that uses a speaker to play a music increases sound intensity, but it may also turn on a small light of the speaker, thus increasing light intensity. An app developer may get confused whether he should specify the effect of increasing the light intensity in this case. However, specifying it will not allow DepSys to detect more conflicts. Also, although we make full use of the *opposite effect* to determine conflicting interventions, the use of the *same effect* is not fully exploited. If two apps turn on two air conditioners to reduce room temperature, DepSys will grant both requests hoping that they will complement each other and stop after reaching to their target setpoints (Section 3.9.3). However, there may be cases when such a strategy can cause loss of comfort, e.g., if multiple apps turn on multiple lights, they may increase room light intensity to an uncomfortable level.

The *emphasis* metadata will not be useful if all the actuations are important to the apps. In a home setting, usually there are actuations that are not important to an app for achieving its goal, e.g., the Discourage Burglar app that turns off lights at 9 PM. However, there are CPS domains where almost all the actuations are important to the apps, e.g., interventions performed by the human-in-the-loop CPS apps. In such cases, *emphasis* will not help to detect false conflicts.

The *condition* metadata only allows specifying primitive conditions based on time. It does not allow specifying conditions based on events, e.g., *App1* will use Light1 when a depression episode is detected. It also does not allow specifying conditions based on complex temporal logic, e.g., *App1* will turn on the AC if the *eventual* temperature reaches 100 degree F, or *App1* will keep the Light1 off *till* the TV is on. Metadata like *eventual* or *till* do not let us know the actual time of the actuation and hence we do not use them.

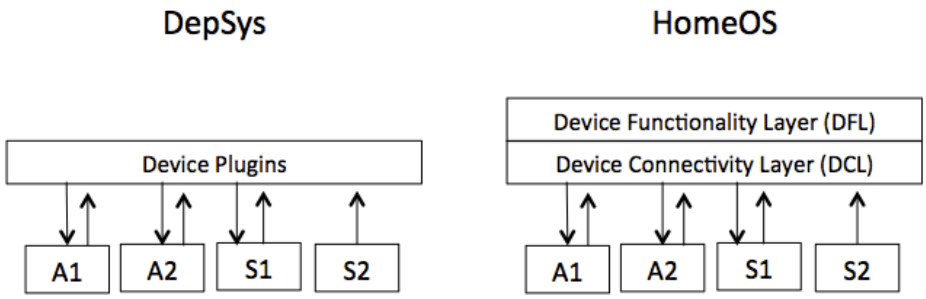


Figure 3.2: Device Plugins layer of DepSys and DCL, DFL of HomeOS

None of the metadata supports specifying dependency information using fuzzy logic, e.g., an app developer can not specify that *App1* will use Light1 with 80% probability. We do not use the probability of conflicts in the dependency analysis. As a result, our installation time dependency checking at the actuator level is very conservative in the sense that if the probability of conflict is 1%, still we assume that there will be conflicts.

3.8 Addressing Functionalities of the Device Plugins Layer

The Device Plugins layer of DepSys implements the ”plug and play” concept, which is not a novel concept and has been used in many systems. We use the lower 2 layers of HomeOS that are called Device Connectivity Layer (DCL) and Device Functionality Layer (DFL) to implement the functionality of the Device Plugins layer as shown in Figure 3.2. We use the functionality of the devices as specified in the DFL to define a vocabulary of conflict detection. So, it is important to briefly describe the DCL and DFL of HomeOS and how these are used in DepSys.

Users can buy COTS devices that use different communication protocols, but have the same functionality. For example, they can buy motion detectors that use X10 and Zwave communication protocols for delivering packets containing motion sensor firing wirelessly. However, to an app, a motion detector is just a motion detector regardless of the underlying protocol it is using. Thus, separating the low level protocols from the functionality of the sensors and actuators frees the app developers from worrying about the low level protocols and it improves extensibility, e.g., a new motion detector can be added that uses a different communication protocol, but still the previously installed apps can use it. This is the main intuition for having two separate layers in HomeOS, DCL and DFL. DCL is protocol dependent while DFL is protocol independent. For this particular example, if the user has X10 and Zwave motion detectors, there will be a driver for X10 and a driver for Zwave in DCL. However, in DFL, both motion detectors will play the role of ”MotionDetector”. DFL exposes device functionality for the apps using a service abstraction in the form of *roles*. Each role contains a list of operations that can be invoked. For example, a ”LightSwitch” role has two operations, ”turnOn” and ”turnOff”, each taking no arguments. Role names are based on device functionality and are unique across device vendors and homes.

Roles	Operations	Effects
LightSwitch	On()	$< light >< increase >$
	Off()	$< light >< decrease >$
Dimmer	On()	$< light >< increase >$
	Off()	$< light >< decrease >$
	ChangeIntensity(arg1)	$< light >< change >$
	GetCurrentIntensity()	
Heater	On()	$< temperature >< increase >$
	Off()	
	ChangeSetpoint(arg1)	$< temperature >< change >$
	GetCurrentSetpoint()	
AC	On()	$< temperature >< decrease >$
	Off()	
	ChangeSetpoint(arg1)	$< temperature >< change >$
	GetCurrentSetpoint()	
HVAC	On()	$< temperature >< change >$
	Off()	
	ChangeSetpoint(arg1)	$< temperature >< change >$
	GetCurrentSetpoint()	
Humidifier	On()	$< humidity >< increase >$
	Off()	
	ChangeSetpoint(arg1)	$< humidity >< change >$
	GetCurrentSetpoint()	
Dehumidifier	On()	$< humidity >< decrease >$
	Off()	
	ChangeSetpoint(arg1)	$< humidity >< change >$
	GetCurrentSetpoint()	
TV	On()	$< light >< change >$ $< acoustic >< change >$
	Off()	
Window	On(), Off(), ChangeSetpoint(arg1)	$< temperature >< change >$ $< light >< change >$ $< acoustic >< change >$ $< humidity >< change >$ $< air\_motion >< change >$ $< odor >< change >$ $< dust >< change >$
	GetCurrentSetpoint()	
Door	On(), Off(), ChangeSetpoint(arg1)	$< temperature >< change >$ $< light >< change >$ $< acoustic >< change >$ $< humidity >< change >$ $< air\_motion >< change >$ $< odor >< change >$ $< dust >< change >$
	GetCurrentSetpoint()	
Speaker	On()	$< acoustic >< increase >$ $< beep >< increase >$
	ChangeVolume(arg1)	$< acoustic >< change >$ $< beep >< change >$
	Off()	

Table 3.1: Roles, operations, and effects of some home appliances

HomeOS comes with a fixed set of roles and operations. New roles can be created by using HomeOS API if needed. We add one additional metadata to each role, which is called *effect* (c.f. Section 3.7.1). The roles, operations, and effects in DFL are part of the conflict vocabulary that DepSys uses to detect runtime conflicts. Some examples of roles, operations, and effects of some appliances in the home are shown in Table 3.1.

Note that the effects are the impact on the environment when the devices are turned on/running, e.g., an AC is on, or a light is on, or a humidifier is on. When a humidifier is on, it increases humidity. So, humidifier effect is $< humidity >< increase >$. Note that when the humidifier is off, the humidity may remain the same or decreases. But that is not the effect of the humidifier device. It would have happened even if the humidifier device were not at the home. The same concept can be applied to the AC and heater. If it is not clear whether the effect-direction is $< increase >$ or $< decrease >$, then just specifying $< change >$ suffices. It will allow DepSys to consider both $< increase >$ and $< decrease >$ behavior at the time of conflict detection, which is a bit pessimistic. Note that the keywords listed in Table 3.1 are part of the conflict vocabulary.

3.9 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.

3.9.1 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 3.4), and if the deployment satisfies the 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 3.6). 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*.

Conflict Semantics at the Installation Time for the Actuators

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 3.2. It shows that if the *conditions* are mutually exclusive, or the *effect* is *different* or the *same*, then these two apps are not conflicting. Otherwise, there is a potential chance of conflict between these two apps.

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

Table 3.2: Truth table for conflict detection

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 3.3. 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	*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 3.3: 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.

3.9.2 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 3.9.1), HomeOS [7] 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

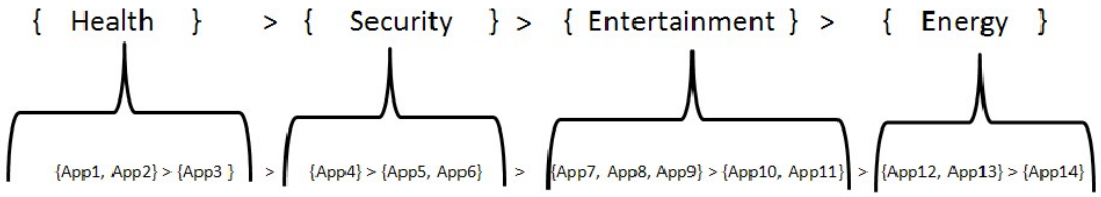


Figure 3.3: Semantic Aware Multilevel Equivalence Class based Policy

residents' speech, and $App3$ plays music based on the environment (e.g., rainy, cloudy, and snowy). It may not matter which app plays 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 3.3. 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 $App1$, 40% time $App2$ and $App3$). Such policies increase the flexibility of the ways apps are run.

3.9.3 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 Checker (SCDC) in DepSys-RC resolves the control dependencies of sensors automatically as described in Section 3.6. When an app wants to control an actuator at runtime, the device control request contains *effect*, *emphasis*, and *condition*. The conflict semantics at the actuator level is specified below.

(a) Runtime Conflict Semantics for the Actuators

There are two cases to consider when the current control request conflicts with a previous control request depending on whether both requests try to control the same device or different devices. We elaborate on both cases in detail.

Case 1: Conflict at the Same Device

When both control requests are targeted for the same device, DepSys-RC uses *emphasis* and device semantics to categorize the conflict into **true conflict** or **false conflict**. The notion of true conflict and false conflict depends on the functionality of the device and the strategies for lights, speakers, and HVAC controllers are described below.

Assume that *App1* and *App2* want to control the same light. DepSys uses the following strategies to determine true and false conflicts involving the light.

- If both apps have emphasis and their operations are different, then it is a true conflict. For example, *App1* wants to turn on the light and *App2* wants to turn off the light. Another example would be, *App1* wants to turn on the light and *App2* wants to change light intensity. This is also a true conflict since *App2* may lower the light intensity so much that violates *App1*'s semantic of turning on the light.

- If both apps have emphasis and their operations are the same, but the difference in parameter values exceeds a defined threshold, then it is a true conflict. For example, *App1* wants to perform ChangeIntensity(arg1) operation on a dimmer while *App2* wants to perform ChangeIntensity(arg2) operation on the same dimmer, then it is a true conflict if $|arg1 - arg2| > T\_dimmer$, where T_dimmer is a threshold. Otherwise, it is a false conflict. Recall that all the allowed operations are defined in the Device Functionality Layer (DFL) of the Device Plugins layer. A default value of the threshold is defined there by the driver developers. However, users can change the threshold if needed.

- If either request does not have emphasis, then it is a false conflict. For example, from table 4.1, when $App\#1$ wants to turn off light at 9PM, this is an optional off request as it does not have emphasis. 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.

The strategy is similar for the HVAC controllers. 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, or when both apps want to adjust the setpoint and the difference of their target setpoints exceeds a defined threshold, these are true conflicts assuming that both requests have emphasis. However, when an app wants to turn on an HVAC controller while another app wants to turn it off optionally (with no emphasis), this is a false conflict. An example of such case would be the turn on event of $App\#7$ and the optional turn off event of $App\#17$ from Table 4.1.

For a speaker, when an app wants to play a beep sound perhaps as an alarm while another app wants to play a music or when two apps want to play music at the same time, these are true conflicts. However, when two apps want to play beep, e.g., $App\#3$ and $App\#5$ from Table 4.1, we consider this a false conflict as the duration of beep is usually very small and the probability of two apps will play a beep at the same time is low.

When there is a true conflict, the Actuator Control Dependency Checker (ACDC) 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 where at least one app does not have emphasis, DepSys just ignores the optional request regardless of app priority. DepSys grants permission to both requests in the remaining cases, e.g., beep requests from two apps to the same speaker, or when two apps want to adjust the setpoint of an HVAC controller and the difference of their target setpoints is within the defined threshold.

Case 2: Conflict at Different Devices

Effect is used to detect conflicts at different devices. If the app developer fails to specify an effect at the runtime "device control request", DepSys uses the effect specified in the Device Functionality Layer associated with the requested operation. Currently, the allowed effects are classified into 2 sets:

- SetA = {light, acoustic, aesthetics}
- SetB = {temperature, radiant_temperature, humidity, dust, odor, air_motion}

If two apps have different effects, then they are not conflicting, e.g., one app is increasing temperature while another app is increasing light intensity. If two apps have opposite effect, e.g., one app is increasing temperature while another app is decreasing temperature, then DepSys checks whether the particular effect belongs to SetA or SetB. If the effect belongs to SetA, then the apps are not conflicting, i.e., if one app is increasing light intensity and another app is decreasing light intensity using different devices, it is not considered a conflict. However, if the effect belongs to SetB, then the apps are conflicting, i.e., if one app is increasing temperature while another app is decreasing temperature using different devices, it is considered a conflict. In case of a conflict, the request of the high priority app is granted and the request of the low priority app is denied. Note that 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.

(b) Addressing Missing Dependencies

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 $App1$ asks to turn on $L1$, 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 3.9.2).

(a) (b)

(c) (d)

Figure 3.4: Aeon Labs Zwave (a) Smart Energy Switch, (b) Multisensor, (c) Door/Window Sensor, and (d) Zstick

3.10 Implementation

The architecture of the DepSys platform along with the designed modules is specified in Section 3.2. Implementing such a huge platform with so many functionalities from scratch requires a tremendous amount of effort. A significant amount of effort is required to develop device drivers. We assume that device vendors develop the drivers of their devices for our platform. However, it is hard to convince the vendors to develop device drivers for academic research. While some wireless protocols are open, e.g., ZigBee, others are proprietary, e.g., Zwave and not open for everyone. In order to save some effort of developing drivers, we leverage the source code of HomeOS [7] that offers Z-wave drivers which enable us to control lights, humidifiers, and dehumidifiers using the Z-wave smart energy switch (Figure 3.4(a)), Z-wave multisensor (Figure 3.4(b)), and Z-wave door/window sensor (Figure 3.4(c)) with a Z-wave USB dongle called ZStick (Figure 3.4(d)) connected to the computer. We develop DepSys upon the HomeOS platform and use the codebase of HomeOS to develop and run smart home apps. We add our proposed metadata to these apps and enhance the the

conflict detection and resolution mechanism of HomeOS by adding the dependency checking proposed by DepSys.

HomeOS is implemented in C# using the .NET 4.0 Framework. HomeOS code is structured like a plugin framework so that a fault in an app or in a driver code does not crash the whole platform. HomeOS treats both apps and drivers as modules and doesn't separate between the two. In order to provide isolation between the platform and the modules and between the modules, HomeOS uses *AppDomains* and *System.AddIn* models. *AppDomains* model allows each module to run in its own application domain and direct manipulation across domains is not allowed. *AppDomains* provides a model for developing plugin framework in .NET and allows independent version control of the platform and the modules so that each can evolve independently.

In this section, we describe the smart home apps that are implemented and tested in the lab, a brief overview of how to write a smart home app, and how we enhance the Kernel of HomeOS to perform sophisticated conflict detection and resolution as proposed by DepSys. Note that all of these depend on the support from the Device Plugins layer. A detailed description of the functionalities of the Device Plugins layer is specified in Section 3.8.

3.10.1 Smart Home Apps Implementation Status

We have implemented and tested 5 smart home apps. A description of the 5 apps are listed in Table 3.4. These apps are currently deployed in the lab.

Serial#	App Name	App Description
1	Discourage Burglar	It turns on all the lights at 8 PM and turns off all the lights at 9 PM. The users can change the turn on and turn off times through a user interface.
2	Motion Based Light Control	It turns on the corresponding light when a motion is detected by a motion detector. It turns off the light after 15 minutes of no motion detection, which is the default timeout interval now. The users can change the timeout interval through a user interface.
3	Lights	Users can control all the lights of the home using this app. They can turn on a single light, turn off a single light, turn on all the lights, and turn off all the lights by just pressing a button in the user interface.
4	Humidifier	Users can turn on and turn off humidifiers using this app.
5	Dehumidifier	Users can turn on and turn off dehumidifiers using this app.

Table 3.4: List of smart home apps that have been implemented

3.10.2 Case Study: Implementation of a Smart Home App

Developing a smart home app is really simple. In this section, we provide an overview on how to write a smart home app. We choose the Discourage Burglar app for this purpose (App #1 from the Table 3.4). An app developer needs to develop a user interface in the the front end and app logic in the back end of the app. We describe both in detail below.

(a) Front End Development

As described above, HomeOS is developed like a plugin framework and there are strategies to provide isolation between the platform and the modules and between the modules. However, when the HomeOS platform runs, it basically runs like a web server. Users install and configure apps using a browser. So, if any user interaction is needed for an app, a

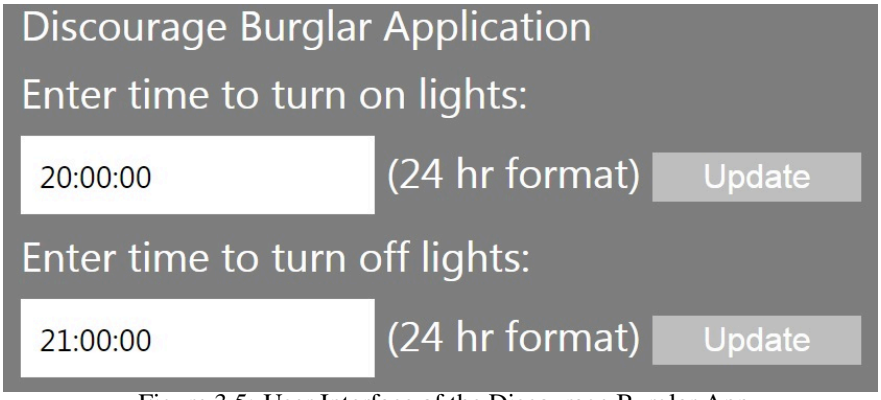


Figure 3.5: User Interface of the Discourage Burglar App

user interface need to be developed for the front end. For our Discourage Burglar app, we need to get user input to determine when to turn on and turn off lights. Hence, we develop a user interface that looks like Figure 3.5.

HTML and JavaScript are used to implement the front end. There are several advantages of using the JavaScript. First, JavaScript is simple and easy to learn. Second, it is versatile, i.e., it can be inserted to any web page. Third, since JavaScript code runs at the client side, it is fast for computation and it reduces the load at the server. When the user interface of the Discourage Burglar app starts, it communicates with the back end of the app and obtains the default turn on time and turn off time from the back end, which are set to 8:00 PM and 9:00 PM in the back end, respectively. Then the JavaScript code displays the default turn on and turn off times on the webpage. The following code is used to achieve this functionality.

```
$(document).ready(
   function () {
      new PlatformServiceHelper().MakeServiceCall("webapp/GetTimeToTurnOn", "", GetTimeToTurnOnCallback);
      new PlatformServiceHelper().MakeServiceCall("webapp/GetTimeToTurnOff", "", GetTimeToTurnOffCallback);
   }
);

function GetTimeToTurnOnCallback(context, result) {
   document.getElementById("textbox_turn_on").value = result.toString();
}

function GetTimeToTurnOffCallback(context, result) {
   document.getElementById("textbox_turn_off").value = result.toString();
}
```

In the above code segment, two service calls are made to the back end of the app: *GetTimeToTurnOn()* and *GetTimeToTurnOff()*. These two functions are implemented in the back end. After the back end returns the value of the *GetTimeToTurnOn()* function call, the JavaScript code gets a callback at function *GetTimeToTurnOnCallback(context, result)*, where it updates the webpage with the default turn on time. Similarly, after the back end returns the value of the *GetTimeToTurnOff()* function call, the JavaScript code gets a callback at function *GetTimeToTurnOffCallback(context, result)*, where it updates the webpage with the default turn off time.

Users can update the turn on and turn off times from the user interface. To update the turn on time, the user provides the updated turn on time in the 24 hour format into the textbox under ”Enter time to turn on lights” and then clicks the ”Update” button at the right side (c.f. Figure 3.5). When the button is clicked, the following JavaScript function $UpdateTurnOnTime()$ is called.

```
function UpdateTurnOnTime() {
   //get the values of the textbox
   var hhmmss = document.getElementById('textbox_turn_on').value;

   var splitted = hhmmss.split(':');
   var hh = parseInt(splitted[0]);
   var mm = parseInt(splitted[1]);
   var ss = parseInt(splitted[2]);

   if (hh < 0 || hh > 23) {
      document.getElementById("label_status").innerHTML = "--Status: Invalid hour";
      return;
   }
   if (mm < 0 || mm > 59) {
      document.getElementById("label_status").innerHTML = "--Status: Invalid minute";
      return;
   }
   if (ss < 0 || ss > 59) {
      document.getElementById("label_status").innerHTML = "--Status: Invalid second";
      return;
   }
   new PlatformServiceHelper().MakeServiceCall("webapp/SetTimeToTurnOn", '{"timeToTurnOn": "' + hhmmss + '"}',
                           SetTimeToTurnOnCallback);
}

function SetTimeToTurnOnCallback(context, result) {
   document.getElementById("label_status").innerHTML = "--Status: Time to turn on is changed to " + result ;
}
```

$UpdateTurnOnTime()$ function first extracts the hour, minute, and second of the updated turn on time from the text box and then performs some check to see if the given time is valid, i.e., if the given hour is between 0 and 23, minute is between 0 and 59, and second is between 0 and 59. If the input is invalid, then the function returns with an error message. Otherwise, the function performs a service call to $SetTimeToTurnOn()$ function, which is implemented in the back end. When $SetTimeToTurnOn()$ returns, $SetTimeToTurnOnCallback(context, result)$ function gets a callback, which shows the user a notification saying that the turn on time has been updated with the given input. To update the turn off time, a similar function $UpdateTurnOffTime()$ is written, which peforms a service call to the $SetTimeToTurnOff()$ function at the back end.

There are some HTML codes in the front end to define the layout of the user interface, i.e., to specify the position and font size of the text boxes and the buttons, as shown below:

```
    <div class="page">
       <div class="row">
          <div class="page_title col">Discourage Burglar Application</div>
       </div>

     <div class="row">
        <div class="page_title col"> Enter time to turn on lights: </div>
        <div class="page_title col">
           <input class="app_form" style="font-size: 22pt" id="textbox_turn_on" type="text" >
          (24 hr format)
          <button class="app_button2" id="button_turn_on" onclick="UpdateTurnOnTime()">Update</button>
        </div>
    </div>

     <div class="row">
        <div class="page_title col"> Enter time to turn off lights: </div>
        <div class="page_title col">
           <input class="app_form" style="font-size: 22pt" id="textbox_turn_off" type="text" >
          (24 hr format)
          <button class="app_button2" id="button_turn_off" onclick="UpdateTurnOffTime()">Update</button>
        </div>
    </div>

  <div class="row">
    <label id="label_status"></label>
  </div>

</body>
```

The code is a very easy to understand for anyone who has the basic knowledge of HTML and hence we omit further explanations.

(b) Back End Development

From the description of the front end development described above, it is clear that the back end of the Discourage Burglar app needs to support the 4 service calls: $SetTimeToTurnOn()$, $SetTimeToTurnOff()$, $GetTimeToTurnOn()$, and $TimeToTurnOff()$. The back end is developed in C#. The service interface that the Discourage Burglar app exposes is defined in C# as follows:

```
  [ServiceContract]
  public interface IDiscourageBurglarSvcContract
  {
     [OperationContract]
     [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.WrappedRequest,
```

```
                                ResponseFormat = WebMessageFormat.Json)]
    string SetTimeToTurnOn(string timeToTurnOn);

    [OperationContract]
    [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.WrappedRequest,
                                ResponseFormat = WebMessageFormat.Json)]
    string SetTimeToTurnOff(string timeToTurnOff);

    [OperationContract]
    [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.WrappedRequest,
                                ResponseFormat = WebMessageFormat.Json)]
    string GetTimeToTurnOn();

    [OperationContract]
    [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.WrappedRequest,
                                ResponseFormat = WebMessageFormat.Json)]
    string GetTimeToTurnOff();
}
```

The service contract $IDiscourageBurglarSvcContract$ mentioned above specifies the operations that the Discourage Burglar Service ($DiscourageBurglarSvc$) supports. An operation can be thought of as a web service method. HTTP POST is used to to submit the request from the web page. $WebMessageBodyStyle.WrappedRequest$ means that both requests and responses are wrapped. JavaScript Object Notation (JSON) format is used to return the response. Discourage Burglar Service ($DiscourageBurglarSvc$) implements the $IDiscourageBurglarSvcContract$ interface as shown below:

```
public class DiscourageBurglarSvc : IDiscourageBurglarSvcContract
{
   protected VLogger logger;
   DiscourageBurglar controller;

   public DiscourageBurglarSvc(VLogger logger, DiscourageBurglar controller)
   {
      this.logger = logger;
      this.controller = controller;
   }
   public string SetTimeToTurnOn(string timeToTurnOn)
   {
      controller.SetTimeToTurnOn(timeToTurnOn);
      return timeToTurnOn;
   }
   public string SetTimeToTurnOff(string timeToTurnOff)
   {
      controller.SetTimeToTurnOff(timeToTurnOff);
      return timeToTurnOff;
   }
   public string GetTimeToTurnOn()
```

```
    {
        return controller.GetTimeToTurnOn();
    }
    public string GetTimeToTurnOff()
    {
        return controller.GetTimeToTurnOff();
    }
}
```

The core application logic stays in the $controller$ object, which is an instance of the $DiscourageBurglar$ class. $DiscourageBurglar$ class inherits from the $HomeOS.Hub.Common.ModuleBase$ class, thus having all the properties of a HomeOS module. When Discourage Burglar app starts, the following $Start()$ function is called, which starts the $DiscourageBurglarSvc$ service and registers for all the ports of interest.

```
public override void Start()
{
    DiscourageBurglarSvc service = new DiscourageBurglarSvc(logger, this);
    serviceHost = new SafeServiceHost(logger, typeof(IDiscourageBurglarSvcContract), service, this,
                                Constants.AjaxSuffix, moduleInfo.BaseURL());
    serviceHost.Open();
    appServer = new WebFileServer(moduleInfo.BinaryDir(), moduleInfo.BaseURL(), logger);

     IList<VPort> allPortsList = GetAllPortsFromPlatform();
     if (allPortsList != null)
     {
        foreach (VPort port in allPortsList)
        {
            PortRegistered(port);
        }
     }
  }
```

Recall that all functionality provided by the drivers of HomeOS is provided via ports. Each port exports one or more services in the form of roles. Each role has a list of operations that can be invoked by an app. For this particular app, it registers to all the ports that has $RoleSwitchBinary$ role. This role is created for the Zwave smart energy switches that are used to turn on and turn off lights. At the final part of the initialization, two timers are started, one for turning on all the lights on the 8 PM and one for turning off all the lights at 9 PM. These are done in the $PortRegistered(port)$ function mentioned above and we skip of the detailed code that implements the function.

When the timer to turn on the lights times out, the following $TimerTimeoutToTurnOn()$ function is called. It adds dependency metadata effects and emphasis in the device control request and turns on all the lights by the $Invoke()$ function shown below. Then the function restarts the timer so that the timer times out again 24 hours later.

```
void TimerTimeoutToTurnOn(object sender, ElapsedEventArgs e)
{
    DateTime current = System.DateTime.Now;
```

```
        ((Timer)sender).Stop();

        //adding effect
        ParamType effect = new ParamType("effect:light,increase;");

        //adding emphasis
        ParamType emphasis = new ParamType("emphasis:on");

        //turning on the light
        ParamType turn_on = new ParamType(true);
        VPort lightPort = TimerToPort[(Timer)sender];
        ParamType app_name = new ParamType(ToString());
        var retVal = Invoke(lightPort, RoleSwitchBinary.Instance, RoleSwitchBinary.OpSetName, turn_on,
                                    effect, emphasis, app_name);
        if (retVal != null && retVal.Count == 1 && retVal[0].Maintype() == (int)ParamType.SimpleType.error)
           System.Console.WriteLine("DB: Error in turning on light: {0}", retVal[0].Value().ToString());

        //Starting the timer to fire 24 hours later
        TimeSpan difference = current.AddDays(1) - System.DateTime.Now;
        int interval = (int)Math.Ceiling(difference.TotalSeconds) * 1000;
        ((Timer)sender).Interval = interval;
        ((Timer)sender).Start();
    }
```

Note that although the data types of effect and emphasis are just strings, they are converted to ParamType data type before calling the $Invoke()$ function. This is because, ParamType data type is used to exchange data between apps and the platform. The timer to turn off lights works the same way. However, in that case, the effect is to decrease light and there is no emphasis on the turn off event.

When $GetTimeToTurnOn()$ function of the $DiscourageBurglarSvc$ service is called, the following function $GetTimeToTurnOn()$ of the $DiscourageBurglar$ class is called eventually.

```
    public string GetTimeToTurnOn()
    {
        string hr = string.Format("{0:00}", start_hr);
        string min = string.Format("{0:00}", start_min);
        string sec = string.Format("{0:00}", start_sec);

        string time = hr + ":" + min + ":" + sec;
        return time;
    }
```

Here, $start\_hr$, $start\_min$, and $start\_sec$ are *int* type member variables of the $DiscourageBurglar$ class capturing the latest preference of the user about when to turn on the lights. Calling $SetTimeToTurnOn(stringtime)$ function updates these member variables and updates the timeout interval of the timer to turn on the lights. Other two functions, $GetTimeToTurnOff()$ and $SetTimeToTurnOff(string)$ are implemented in a similar fashion.

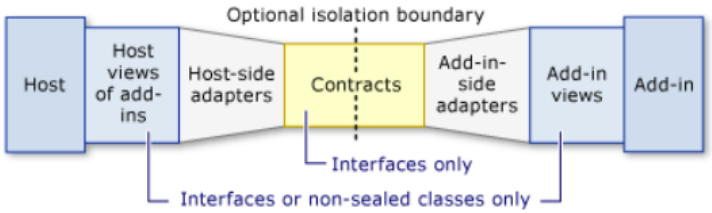


Figure 3.6: The Add-in Pipeline. This figure is obtained from Microsoft Developer Network [2].

3.10.3 Implementation of Conflict Detection and Resolution Techniques in the Platform

In this section, we describe how we implement the runtime conflict detection and resolution techniques described in Section 3.9.3 into the HomeOS platform. The implementation is done in three steps.

First, we intercept each app's device control request before the request is passed to the device driver. Since each app is treated as a *module* in HomeOS, each app needs to inherit from the *HomeOS.Hub.Common.ModuleBase* class. In order to control an appliance, e.g., a light or a humidifier, an app needs to call the $Invoke()$ function in the *ModuleBase* class. So, we intercept each app's control request at the $Invoke()$ function and change the function so that the dependency meta data provided by the app is passed to a function for DepSys's dependency checking before the control request is granted. Since the dependency checking is performed in the platform and HomeOS maintains an isolation between the apps and the platform, passing the dependency metadata from the apps to the platform becomes difficult, which is addressed in the next step.

Second, in order to make sure a fault in an app does not cause the platform to crash, homeOS uses *Add-in* model to provide isolation between the apps and the platform. The Add-in model consist of a series of segments that builds the add-in pipeline [2], which is shown in Figure 3.6. To match the figure with our context, assume that the Host represents the platform and the Add-in represents an app. The figure shows that the Add-in model provides a symmetric communication model to exchange data between the add-in (app) and the host (platform). The Add-in model suggests to keep an optional isolation boundary between the platform and the app. A detailed description of how the add-in model works is specified in Microsoft Developer Network (MSDN) [2]. According to the MSDN [2], the contract segment in the middle of the pipeline is loaded into both the platform's application domain and the app's application domain. The contract defines the virtual methods that the platform and the app use to exchange types with each other. To pass through the isolation boundary, types must be either contracts or serializable types. Types that are not contracts or serializable types must be converted to contracts by the adapter segments in the pipeline. The view segments of the pipeline are abstract base classes or interfaces that provide the platform and the app with a view of the methods that they share, as defined by the contract. We extend the contracts and adapters of HomeOS to exchange data between the apps and the dependency checker at the platform. To be more precise, we write functions to convert dependency checking related parameters from view to contracts and contracts to view in classes in $HomeOS.Hub.Platform.Adapters$

and $HomeOS.Hub.Platform.Views$.

Third, after the dependency metadata is passed to the DepSys runtime dependency checker at the platform, the runtime dependency checker checks for possible conflicts. It maintains a queue containing all the active requests from all the apps running in the home. It compares the current request with the requests in the queue for potential conflicts using the semantics of runtime conflicts specified in Section 3.9.3. If current request does not conflict with any other requests in the queue, the queue is updated with the new request and the request is granted to be passed to the device driver. If a conflict is detected, the runtime dependency checker uses the policy specified in Section 3.9.3 to resolve the conflict. The conflict resolution may grant or deny the new request, or it may grant the new request, but deny a previous request in the queue. The queue is updated accordingly to hold all the active requests.

3.11 Discussion

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.

We do not address actuator level conflicts due to secondary dependencies. For example, an app that turns on a light bulb increases light intensity, which is its primary effect. But it also increases the room temperature a bit, which is the secondary effect that may conflict with an actuation targeted for reducing the room temperature. Such secondary dependency issues are usually minor in a home setting and hence we ignore these. However, it can be addressed by asking the app developers to specify all the secondary effects, or by using a model of the environment that automatically computes all the secondary effects of each actuation. Although we use neither strategy to detect actuator level conflicts due to secondary dependencies, we use the latter strategy to detect conflicting interventions of the apps involving human-in-the-loop (Chapter 6).

Our proposed metadata, *effect*, *emphasis*, and *condition* are powerful enough to detect conflicting actuations that occur within and across devices. However, they have a few limitations. The limitations are described in Section 3.7.4.

DepSys is designed for non-safety critical systems. Although we detect conflicting interventions by taking into account a wide range of physiological parameters of the user and warn the user when a conflicting intervention is detected (Chapter 6), 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.

DepSys does not take any attempt to maximize the number of satisfied control requests. It does not ask the apps to send their planned control requests in advance so that DepSys can schedule the requests. Instead, control requests are processed in the order they come to DepSys. DepSys does not deny a request by anticipating a future request or to maximize the number of concurrently running apps.

3.12 Summary

This chapter provides an overview of the design and implementation of DepSys. We discuss the spectrum of dependencies that DepSys addresses and the way it addresses various dependencies including requirement, name, missing, and control dependencies. We describe the dependency metadata that the app developers are suggested to specify, the role of the user, and the installation and runtime checking performed by DepSys for addressing the actuator control dependency. We also describe how we implement DepSys upon the HomeOS platform and provide a brief overview of how to write a smart home app in this chapter.

Chapter 4

Actuator Control Dependencies Evaluation

In this chapter, we evaluate DepSys in terms of its ability to address control dependencies at the actuator level. Actuator Control Dependency Checker in DepSys (shown in Figure 4.1) addresses control dependencies at the actuator level by doing dependency checking at the app installation time and at runtime. To do that, it uses novel metadata *effect*, *emphasis*, and *condition* as described in Chapter 3.7. We demonstrate the severity of actuator level dependencies that can arise in a home setting by using 219 days of real-home data and the ability of DepSys in detecting and resolving these dependencies in this chapter. We compare DepSys's performance with that of the state of the art solution HomeOS [7] in terms of their abilities to detect and resolve control dependencies at the actuator level. We also compare the overhead of the users in using DepSys and using HomeOS. Finally, we assess the effort of the app developers in specifying our suggested dependency metadata to detect and resolve actuator level dependencies. DepSys's performance to address other dependencies, i.e., sensor reliability dependencies, human-in-the-loop dependencies, and dependencies due to real-time constraints are evaluated in Chapters 5, 6, and 7, respectively.

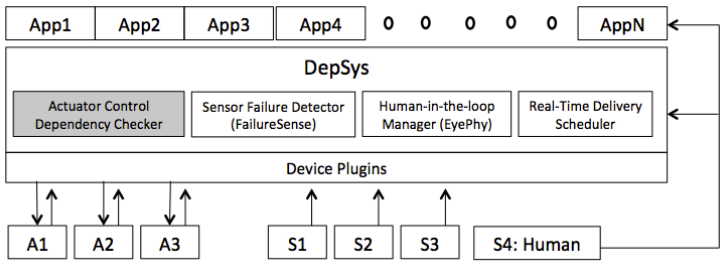


Figure 4.1: Actuator Control Dependency Checker in DepSys

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 4.1: Our App Store containing 35 apps

4.1 App Selection

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 [106] and Google play [107], the apps in these app stores are mainly limited to smart phones and tablets.

There is no such well-established app store for the home to date. Hence, 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 4.1. 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 1.3 million and 1.2 million apps in Android's and Apple's app store, respectively [102] [103], 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.

4.2 Static Analysis

We assume that the sensor/actuator deployment in the home supports all the requirements of the 35 apps in Table 4.1. Before running an app, DepSys-SC performs static analysis at installation time by analyzing the dependency information specified in the app metadata. Figure 4.2 shows the probability of true conflict between at least j apps when i apps are installed from the 35 apps in Table 4.1. 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 4.2. Figure 4.2 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

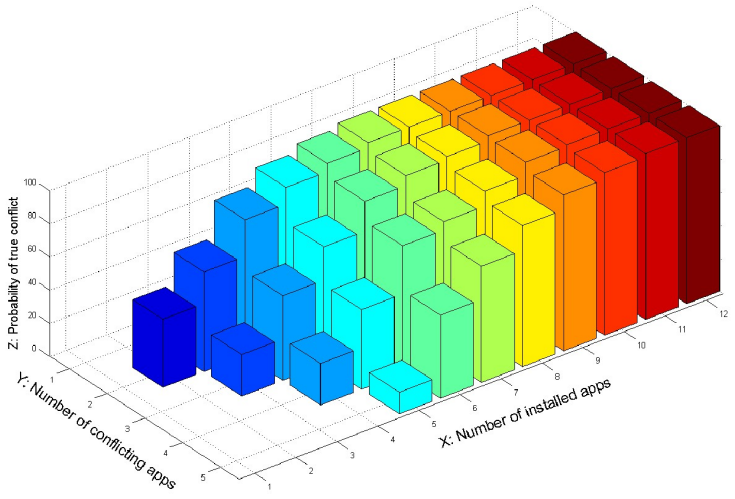


Figure 4.2: Static analysis of conflicts among apps

home setting and demonstrate the need for detecting and resolving runtime conflicts.

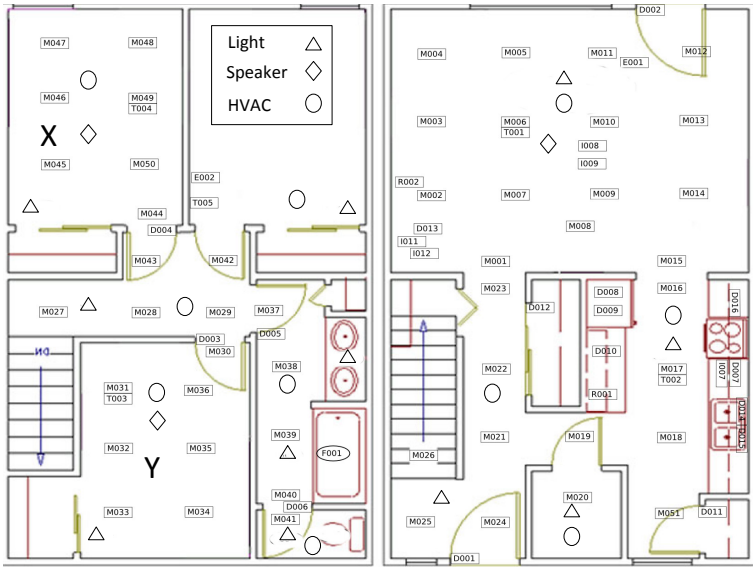


Figure 4.3: Floorplan and position of sensors and actuators

4.3 Runtime Analysis

We use a dataset collected from the WSU CASAS smart home project [10] for performing runtime analysis (dataset name: *twor.2010*). The data was collected between August 24th, 2009 and April 30th, 2010 for a period of 250 days. However, there are 31 days within that period when no sensor firings are reported. We ignore these days and consider the remaining 219 days of data for the evaluation. We split the 219 days into 8 months and the mapping between the days of the months and the actual dates of the data collection is shown in Table 4.2. For example, day 1 of month 1's data is the data collected on August 26th 2009, day 2 of month 1's data is the data collected on August 27th 2009, day 30 of month 1's data is the data collected on September 22nd 2009, and so on. This mapping can be used with the later results to understand if there is any pattern of device level conflicts in different seasons/months of the year.

Month#	Days	Dates of data collection
1	1 . . . 30	Aug 26 - Sept 22 (2009)
2	1 . . . 30	Sept 23 - Oct 22 (2009)
3	1 . . . 30	Oct 23 - Oct 25, Nov 2 - Nov 28 (2009)
4	1 . . . 30	Nov 29 - Dec 9, Dec 18 - Dec 23, Dec 29 (2009), Jan 2, Jan 7 - Jan 17 (2010)
5	1 . . . 30	Jan 18 - Feb 16 (2010)
6	1 . . . 30	Feb 17 - March 12, March 14, March 18 - March 22 (2010)
7	1 . . . 30	March 23 - April 21 (2010)
8	1 . . . 9	April 22 - April 30 (2010)

Table 4.2: Mapping between different days of months and actual dates of data collection

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 4.3.

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 would have controlled the lights, HVAC, and speakers of the testbed if they were actually there. 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 meals, entering the 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.

4.3.1 Number of run-time conflicts

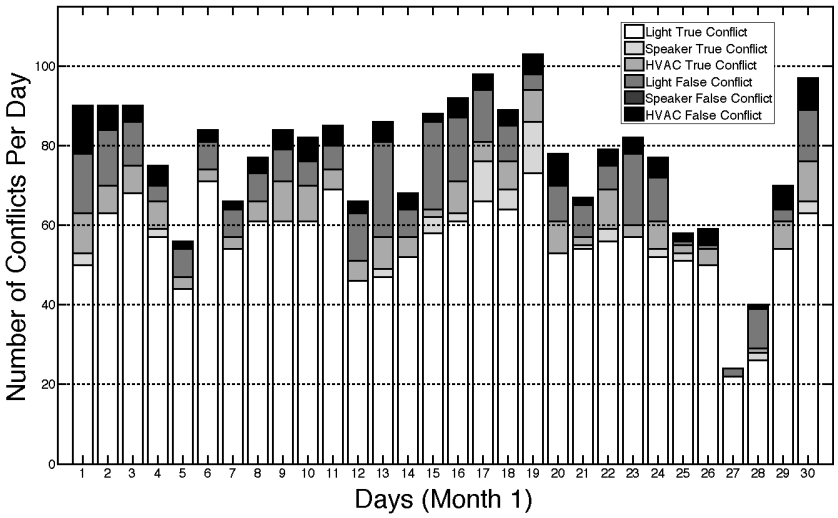


Figure 4.4: Number of conflicts at different days of month 1

Figures 4.4 to 4.11 show the number of conflicts on different days of different months 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 Figures 4.4 to 4.11 that more conflicts occur with the lights than with the speakers and HVAC. No false conflicts are detected on the speakers. On average 70.21 conflicts take place per day, which is really alarming. Among these 70.21 conflicts, 12.06 conflicts are false conflicts, which is 17.18% of the total conflicts. It can be translated to a significant number if people install hundreds of apps in their homes. It shows a great opportunity to avoid these false conflicts if the system is capable of separating the false conflicts from the true conflicts.

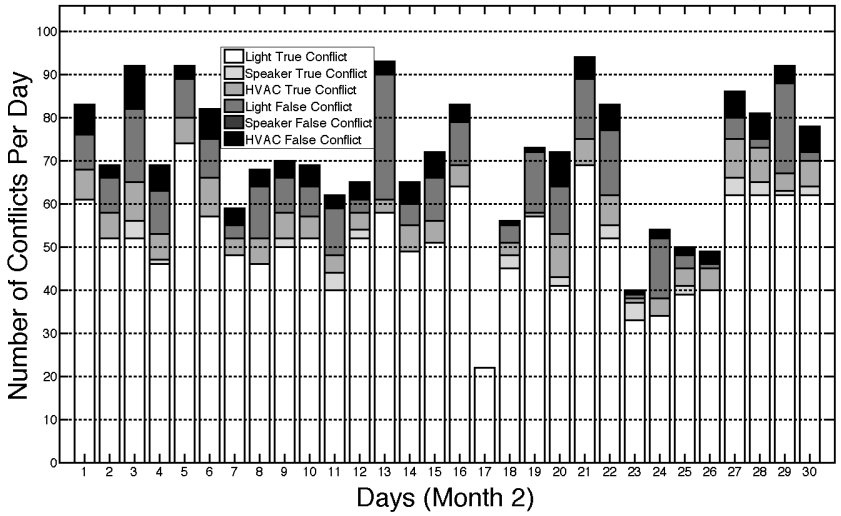


Figure 4.5: Number of conflicts at different days of month 2

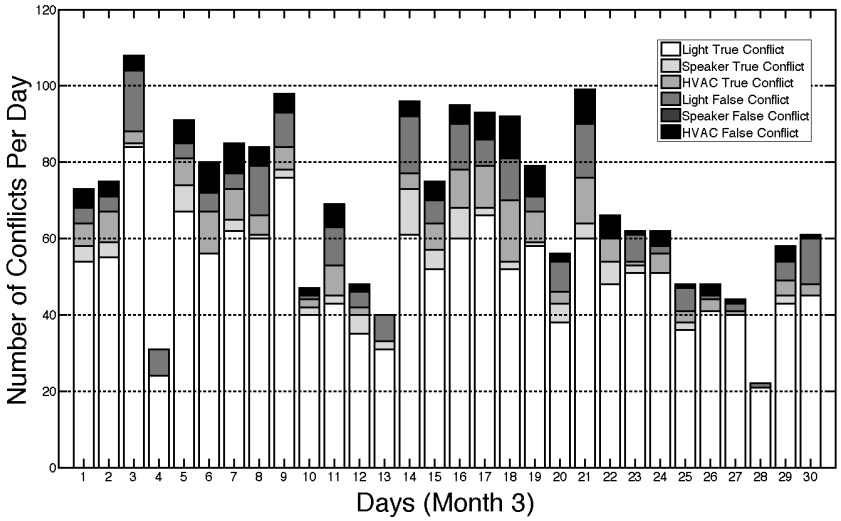


Figure 4.6: Number of conflicts at different days in month 3

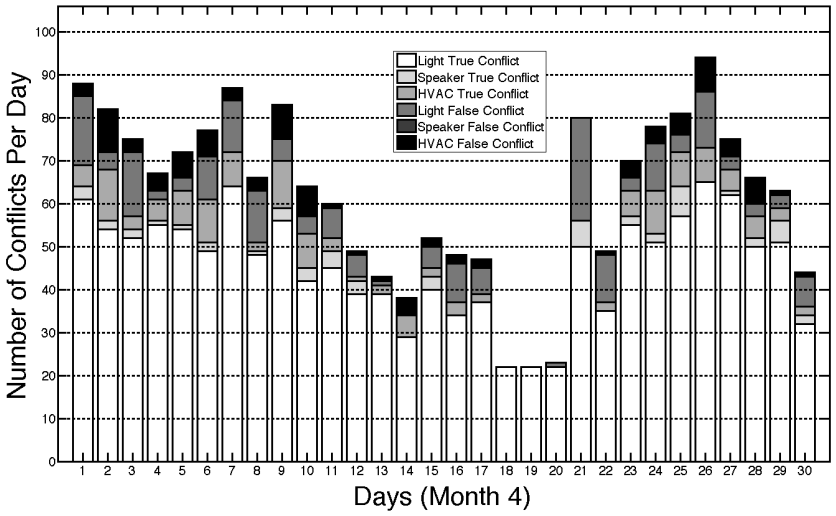


Figure 4.7: Number of conflicts at different days of month 4

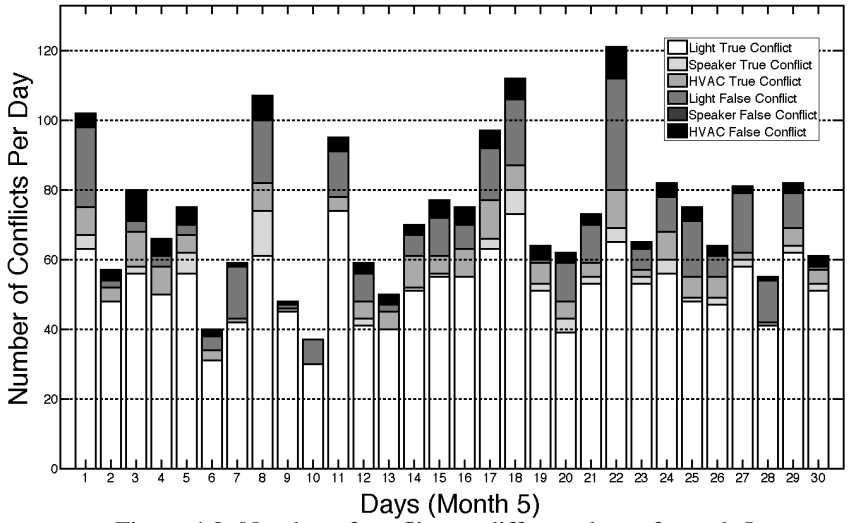


Figure 4.8: Number of conflicts at different days of month 5

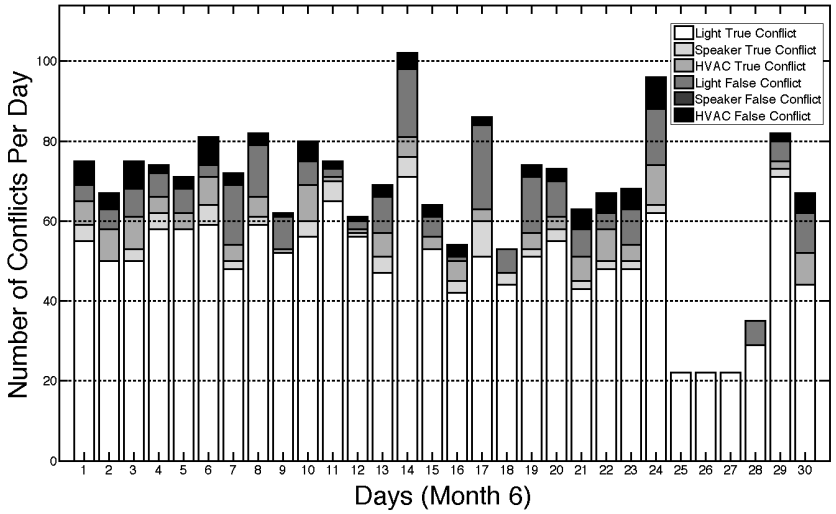


Figure 4.9: Number of conflicts at different days of month 6

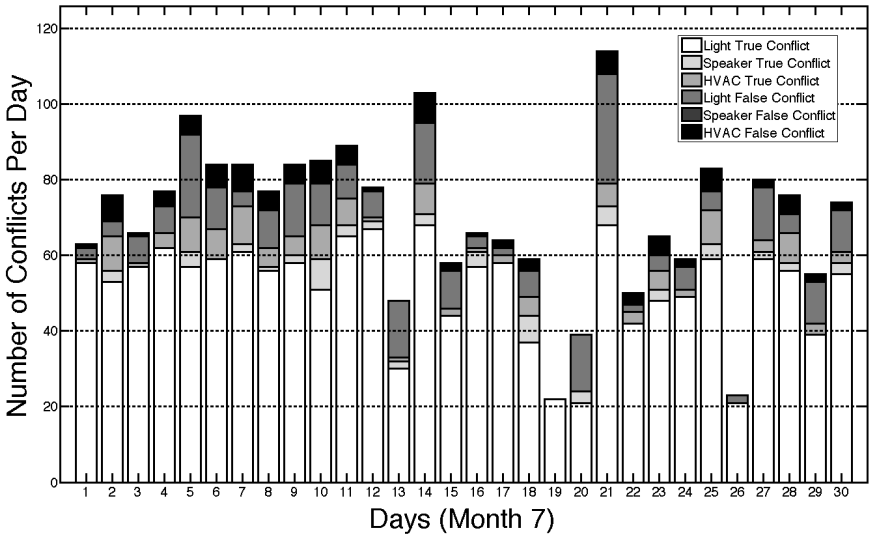


Figure 4.10: Number of conflicts at different days of month 7

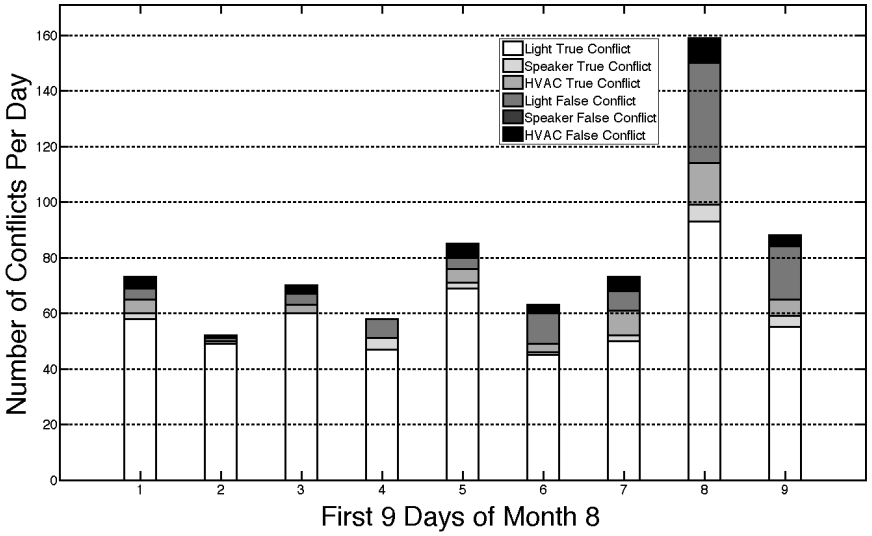


Figure 4.11: Number of conflicts at the first 9 days of month 8. These 9 days are the last 9 days of the 219 days of the data collection.

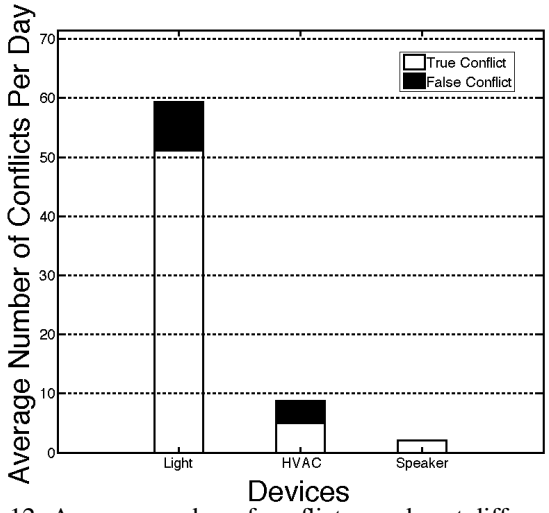


Figure 4.12: Average number of conflicts per day at different devices

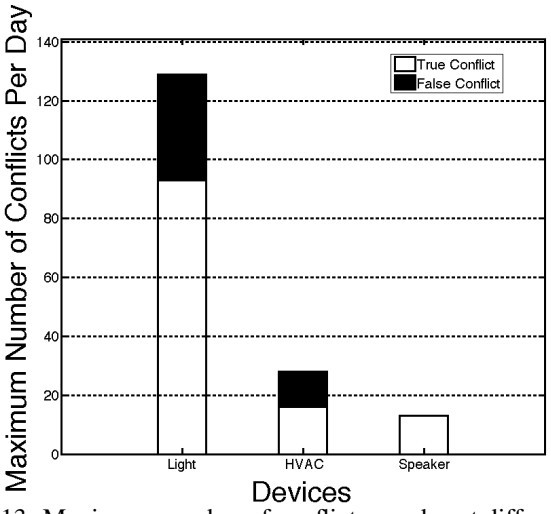


Figure 4.13: Maximum number of conflicts per day at different devices

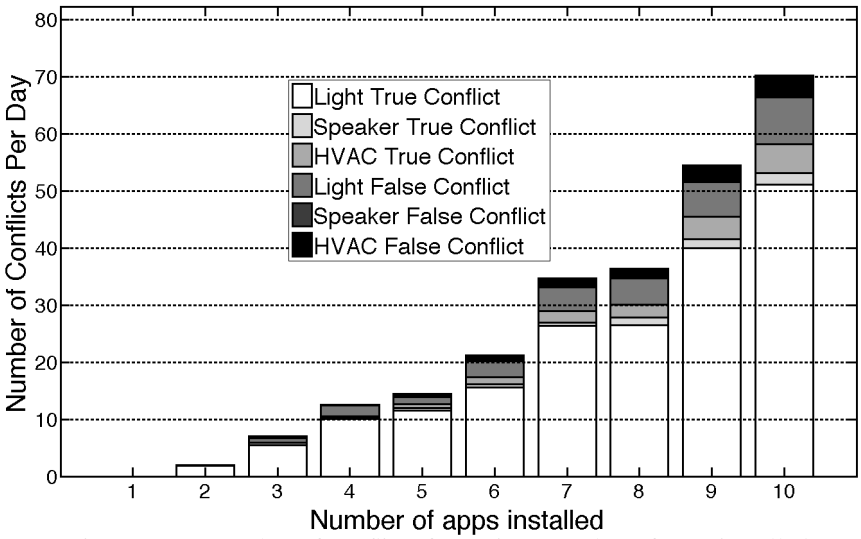


Figure 4.14: Number of conflicts for various number of apps installed

Figures 4.12 and 4.13 show the average and maximum number of conflicts per day for lights, HVAC controllers, and speakers from the 219 days of data. We see from Figure 4.12 that on average 51.10 true conflicts and 8.28 false conflicts are observed for lights per day. For HVAC controllers, on average 5 true conflicts and 3.79 false conflicts are observed per day. For speakers, 2.05 true conflicts are observed per day and no false conflicts are observed. However, from Figure 4.13 we see that maximum 93 true conflicts and 36 false conflicts are observed per day for lights. For HVAC controllers, a maximum of 16 true conflicts and 12 false conflicts are observed per day. For speakers, a maximum of 13 true conflicts are observed per day. We see that the number of conflicts for the lights are much more than that of the HVAC controllers and the speakers. The reason is that there are 11 lights in the home and 7 out of 10 apps control the lights based on the behaviors of the 2 residents. In case of HVAC, there are 10 HVAC controllers and only 2 apps control the HVAC controllers. In the case of the speakers, there are 3 speakers and only 4 apps control the speakers. The number of conflicts on a type of device, e.g., lights, HVAC controllers, and speakers depends on the number of instances of the device, number of installed apps controlling the device, the logic of these apps, the number of residents of the house, and their behaviors.

Since the number of conflicts depends on the number of apps installed, to investigate this further 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 determine the number of conflicts per day by using the CASAS dataset and present the average results in Figure 4.14. The magnitude of conflicts we see in Figure 4.14 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% of the conflicts remain false conflicts on average, which can be a significant number when people install hundreds of apps. Considering more than 1.3 million and 1.2 million apps in Android's and Apple's app stores to date, respectively [102] [103], and since average Android and iPhone users have 32 and 44 apps downloaded in their smart phones, respectively [108], people can install a lot more

apps in a home setting, as there are far more devices in a home than that of a smart phone.

4.3.2 Conflict resolution capability

In this section, we compare the conflict resolution capability of DepSys with that of a state of the art solution, HomeOS [7]. DepSys is more powerful in resolving conflicts because it considers device semantics and can separate false conflicts from true conflicts (c.f. Section 3.9.3). 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 3.9.3). 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 the 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, respectively. 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.

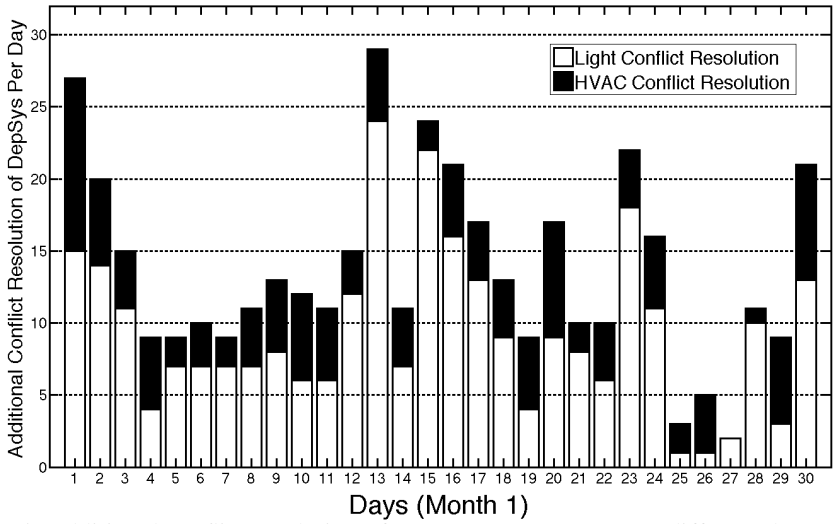


Figure 4.15: Additional conflict resolution of DepSys over HomeOS at different days of month 1

Figures 4.15 to 4.22 show the number of additional conflict resolutions of DepSys per day at different months 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 11.95 on average, it

can be as high as 45 in a day, some of which can be potential discomfortable events. This many conflicts can be very annoying if home appliances are mistakenly turned on/off 45 times in a day. It can be even more if tens or hundreds of apps are installed and running in the home. These results show the conflict resolution capability of DepSys over HomeOS, as none of such conflicts can be resolved accurately by HomeOS.

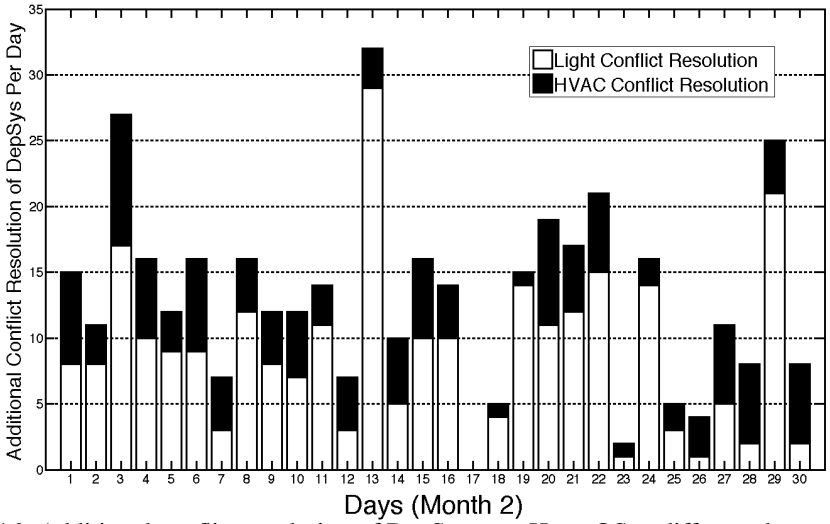


Figure 4.16: Additional conflict resolution of DepSys over HomeOS at different days of month 2

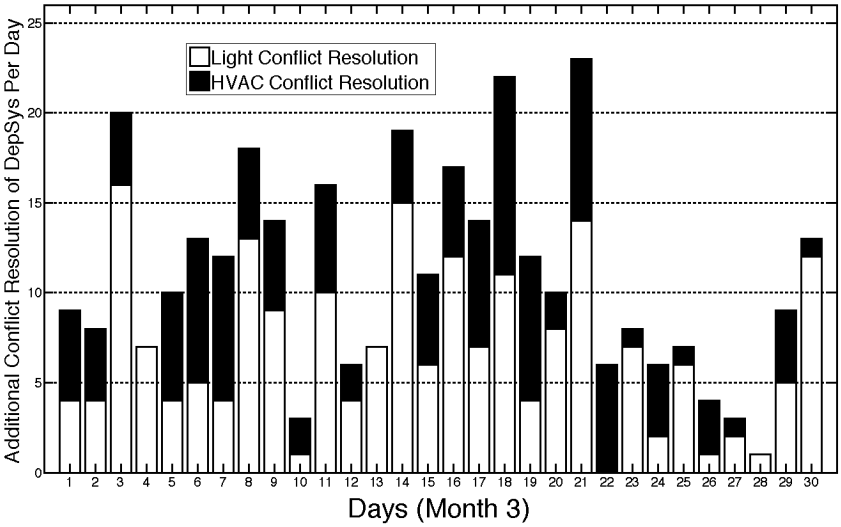


Figure 4.17: Additional conflict resolution of DepSys over HomeOS at different days of month 3

4.3.3 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

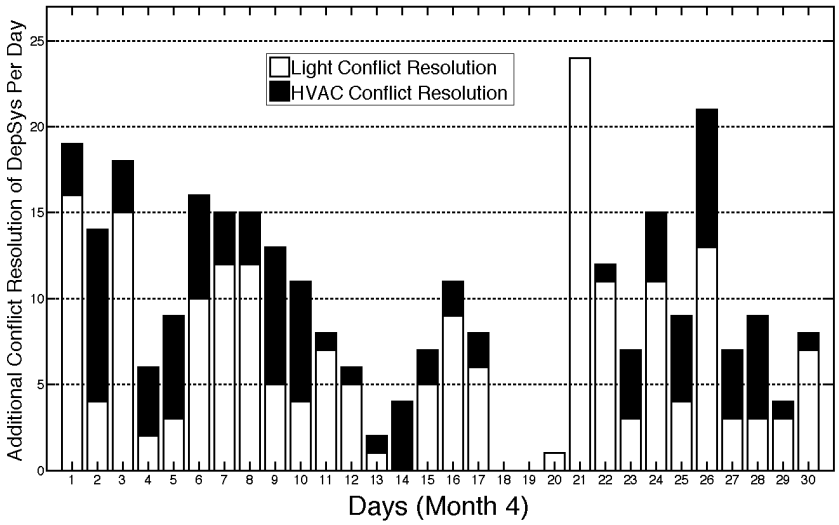


Figure 4.18: Additional conflict resolution of DepSys over HomeOS at different days of month 4

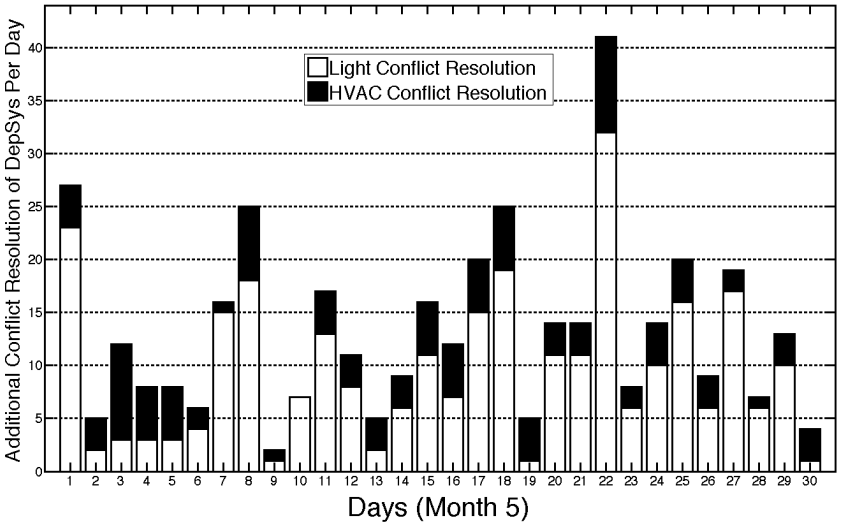


Figure 4.19: Additional conflict resolution of DepSys over HomeOS at different days of month 5

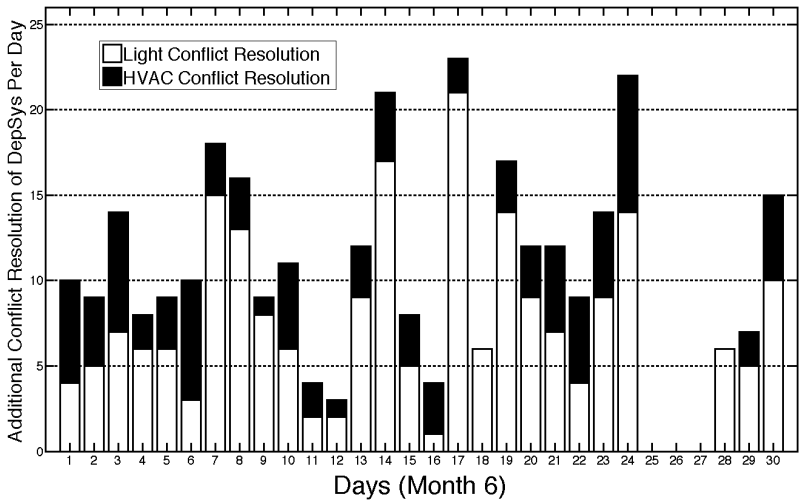


Figure 4.20: Additional conflict resolution of DepSys over HomeOS at different days of month 6

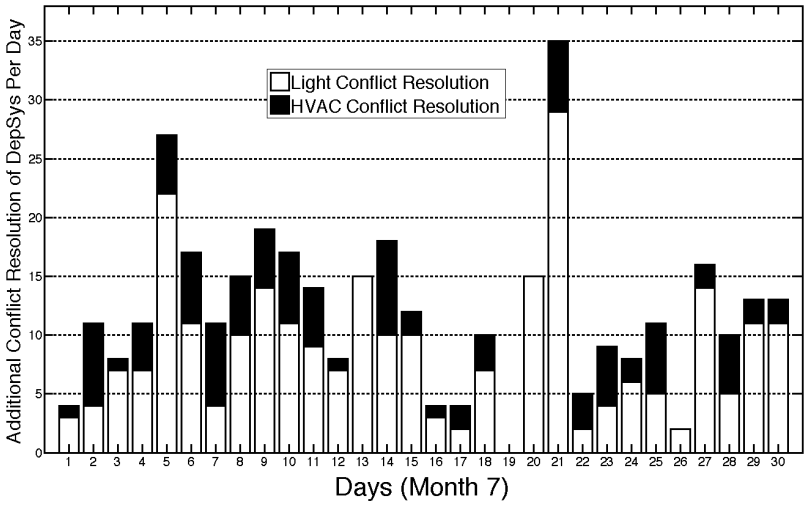


Figure 4.21: Additional conflict resolution of DepSys over HomeOS at different days of month 7

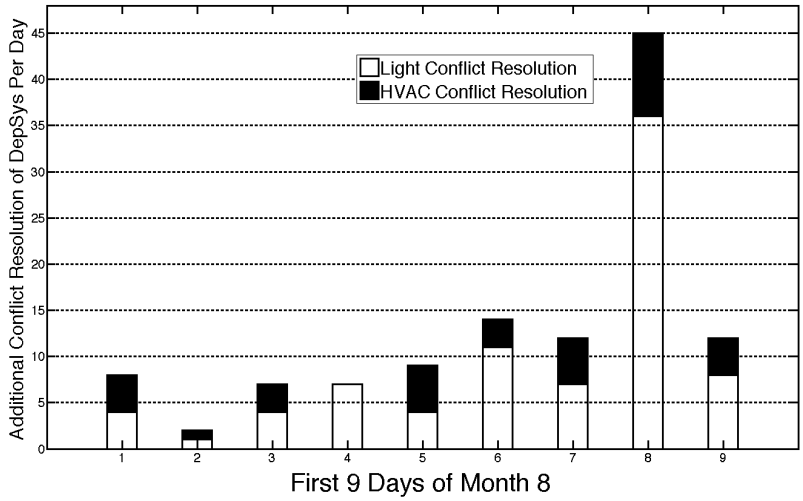


Figure 4.22: Additional conflict resolution of DepSys over HomeOS at the first 9 days of month 8. These 9 days are the last 9 days of the 219 days of the data collection.

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 4.23. 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 4.23 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.

4.4 Overhead for Users

The above evaluation shows the ability of DepSys to detect and resolve actuator level conflicts accurately. However, it is important to assess if this gain is at the cost of increased efforts of users. It is very difficult to measure user effort since it depends on the type of the user, his age, his interaction capability with the computer, the designed user interface,

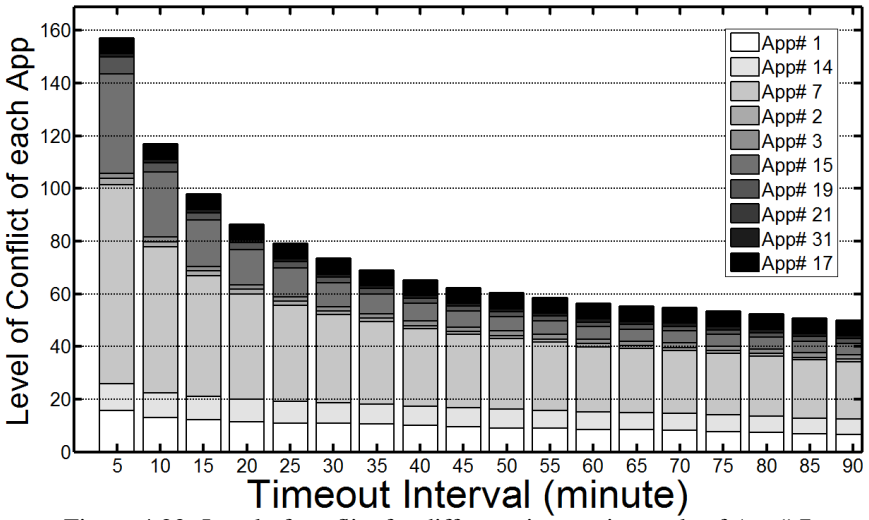


Figure 4.23: Level of conflict for different timeout intervals of App# 7

the apps he is planning to install in his home, and his preferences of conflict resolution. The efforts of a single user may even change over time after he gets used to using the system. A large user study needs to be performed in order to evaluate the way users perceive their effort, which is difficult to do with the available resources. In this thesis, we objectively compare user efforts required by DepSys with that of HomeOS without going through a large user study. The metric of evaluating user effort is *the number of apps that need to be compared* when a new app is installed at the app installation time. To be more precise, assume that a user has already installed N apps and he is installing a new app $AppM$. We are interested to know how many apps among the previously installed N apps the user needs to compare to specify the conflict resolution policy, i.e., priority of the new app $AppM$ at app installation time. He may need to compare with all the N apps or only a few of them depending on the strategy employed by the underlying smart home platform. We evaluate the performance of four strategies:

1. **HomeOS:** When a new app $AppM$ is installed, HomeOS determines the previously installed apps that use the same device as $AppM$ specifies in the app installation metadata. Assume that there are K apps among the N apps that use the same device as $AppM$ has specified. Then the user is asked to provide the priority of $AppM$ in order to maintain a total order among the $K + 1$ apps (including $AppM$).

2. **Using Emphasis:** Recall from Section 3.7.2 that when two apps specify the *same* emphasis about using a device in the app installation metadata, they do not conflict at runtime even though they plan to use the same device. For example, among the 35 apps mentioned in the Table 4.1, App #1, #5, #6, #14, #18, #26, #27, and #28 have the emphasis to turn on the lights. At runtime, we can turn on the requested light and satisfy all of them. So, even though the apps plan to use the same device, they will not conflict at runtime and no user input is necessary to resolve conflicts among these apps. Similarly, for the HVAC system, App #10, #17, and #18 have the same emphasis to turn on the HVAC system. For the speakers, we assume that two apps will not conflict at runtime

even though they plan to use the same speaker when they plan to play small beep sounds to alert the residents, since the probability of two beeps at the same time is very low and we haven't observed any such events during the 219 days of data collection. So, App #3, #5, #9, #20, #25, #26, #27, and #28 from Table 4.1 will not conflict on using the speaker and hence no user input is needed to resolve priorities among these apps regarding their usage of the speakers. These metadata are specified by the app developers. These metadata enable understanding the usage of the appliances at a deeper level and reduces the comparison efforts of the users.

3. **Using SAMECP:** Recall from Section 3.9.2 that Semantic Aware Multilevel Equivalence Class based Policy (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. However, if there are other apps in the same group that use the same device as $AppM$ is intending to use, then the user needs to specify priority of $AppM$ to maintain a total order among these apps. Note that SAMECP allows creating equivalence classes of apps where no app priority is needed within an equivalence class, which improves the flexibility of the ways apps can be run. However, we are not using this equivalence class feature in this evaluation so that HomeOS and our solution can achieve the same policy in terms of app priorities.

4. **Using Emphasis and SAMECP:** This situation combines the strategies specified above in (2) and (3).

We use all the 35 apps in Table 4.1 in this evaluation. We need to make sure that all the four strategies achieve the same app priority order after the user feedback. The app priorities to be achieved are the following: Health apps >Security apps >Entertainment apps >Energy apps. Within Health apps, App #22 >#20 >#19 >#18 >#21. Within Security apps, App #4 >#2 >#3 >#5 >#6 >#1. Within Entertainment apps, App #26 >#27 >#28 >#29 >#30 >#31 >#23 >#24 >#25 >#32 >#33 >#34 >#35. Within Energy apps, App #16 >#13 >#11 >#12 >#10 >#14 >#17 >#9 >#15 >#8 >#7.

We use two metrics to measure the efforts of the users:

(a) **ALL_CONFLICTING:** When a user installs $AppM$, assume that a strategy, e.g., HomeOS/SAMECP decides that $AppM$ may conflict with the previously installed K apps. Note that for a fixed set of N previously installed apps, K can be different by different strategies. When we use this metric, the user effort is estimated as K as it assumes that the user has to compare with all the K potentially conflicting apps before deciding the priority of $AppM$, which may be the case for some users. This metric captures the users' efforts in a more pessimistic way than the next metric *MIN_CONFLICTING*.

(b) **MIN_CONFLICTING:** This metric is based on the insight that the user does not need to compare with all the K potentially conflicting apps to decide the priority of $AppM$ since the K apps are already sorted. When the new app $AppM$ has the highest priority or the lowest priority, then the user does not need to look at any of the K apps and hence the effort is estimated as 0. If the $AppM$ has the second highest priority or the second lowest priority, then the user just needs to look at the highest priority app or the lowest priority app and therefore the effort is estimated as 1. To generalize the strategy, assume that i apps precede $AppM$ and j apps follow $AppM$ in the sorted order of $AppM$ and K apps when we sort these apps using app priorities. *MIN_CONFLICTING* metric suggests to pick the minimum of i and j as the estimated comparison effort of the user. Thus, the user just needs to compare with the minimum number of apps that either precede $AppM$ or follow $AppM$ in the sorted order of apps instead of comparing with all the K apps.

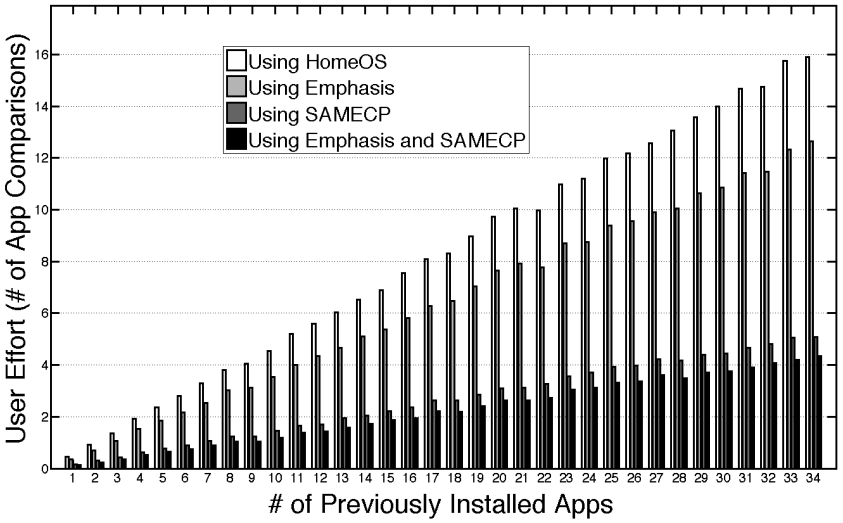


Figure 4.24: User effort measured by *ALL_CONFLICTING* metric using different strategies

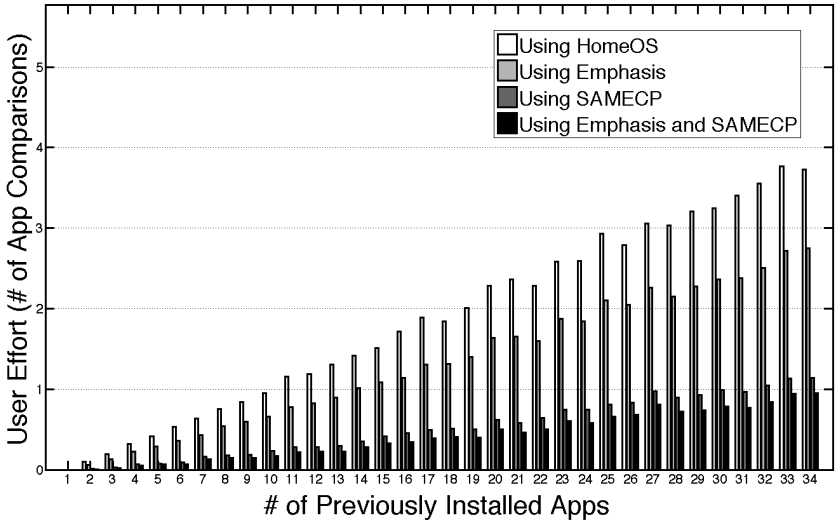


Figure 4.25: User effort measured by *MIN_CONFLICTING* metric using different strategies

When a user installs a new app, his comparison effort depends on the previously installed apps. If no apps are previously installed, then the estimated comparison effort is 0 regardless of the strategy and metric we use. We consider the number of previously installed apps from 1 to 34 in this evaluation and show the user effort in Figures 4.24 and 4.25 using metrics *ALL_CONFLICTING* and *MIN_CONFLICTING*, respectively. For each value x in the X axis, we randomly choose (x + 1) apps 1000 times, treating one app as the new app and the other x apps as the previously installed apps and show the average user comparison effort from the 1000 runs in both figures. We see that user effort increases with the increase of previously installed apps from the both figures. We also see that user effort is maximum when using HomeOS. To quantify the reduction of user effort in DepSys that uses emphasis and SAMECP compared to HomeOS, user effort is averaged over all the (1-34) previously installed apps. We see that the average number of apps comparison required by HomeOS is 8.21 whereas it is 6.41, 2.64, and 2.22 by using emphasis, SAMECP, and DepSys, respectively using the metric *ALL_CONFLICTING*. Hence, emphasis, SAMECP, and DepSys reduce user effort by 21.92%, 67.84%, and 72.96%, respectively compared to HomeOS. When we use *MIN_CONFLICTING* metric, the average number of apps comparison needed by HomeOS is 1.87 while it is 1.33, 0.52, and 0.42 by emphasis, SAMECP, and DepSys, respectively. Thus, emphasis, SAMECP, and DepSys reduce user effort by 28.88%, 72.19%, and 77.54%, respectively compared to HomeOS. Thus, DepSys not only detects and resolves a significant number of actuator level conflicts compared to HomeOS, it does so with much less user effort.

4.5 Effort of the App Developers

Although DepSys detects and resolves a significant number of control dependencies at the actuator level, DepSys's improved capabilities 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. Because, the app developers need to write only 25.17 lines of dependency metadata per app on average, considering the 35 apps specified in Table 4.1. Hence, DepSys requires minimal effort from the app developers to specify dependency metadata.

4.6 Summary

This chapter describes the performance of DepSys in terms of its ability to address actuator control dependency. By using 219 days of data collected from a real home, we estimate an average of 70.21 conflicts taking place per day in a smart home with only 10 apps runring, which demonstrates the magnitude of runtime conflicts in a home setting. Comparing with the state of the art solution HomeOS, the additional conflict resolution of DepSys per day is on average 11.95, which can be as high as 45 a day. This improved conflict resolution capability of DepSys does not even require any additional user effort. Instead, DepSys reduces user effort by 77.54% compared to HomeOS. This capability comes

at the cost of app developers' efforts to specify additional metadata containing the dependency information. However, their effort is minimal as they only need to write on average 25.17 lines of code per app.

Chapter 5

Addressing Dependencies on Sensor Reliability Status

One of the major motivations for integrating Cyber-Physical Systems (CPSs) in a smart home is to allow the CPS apps to share the underlying sensors. If all the apps share a single set of sensors, it will reduce the cost of sensor deployment, increase the aesthetics of the rooms, and reduce channel contention for packet transmission. However, if a single sensor fails, that will affect the performance of all the dependent apps. Because, apps have dependencies on the reliability status of the sensors that they are using. In this chapter, we describe how DepSys addresses such sensor reliability dependencies. "Sensor Failure Detector" component in DepSys (shown in Figure 5.1) runs a novel technique called FailureSense to detect realistic sensor failures in a smart home and reports it to the dependent apps when a failure is detected. This chapter presents the detailed design and evaluation of the FailureSense scheme.

The rest of the chapter is organized as follows. Section 5.1 discusses the motivation of using FailureSense. Section 5.2 lists the technical contributions that this chapter makes. Section 5.3 describes the FailureSense solution in detailed. Section 5.4 and 5.5 describe performance evaluation of FailureSense and its comparison with the state of the art, respectively. Section 5.6 discusses some aspects of the solution and Section 5.7 provides a summary of the chapter.

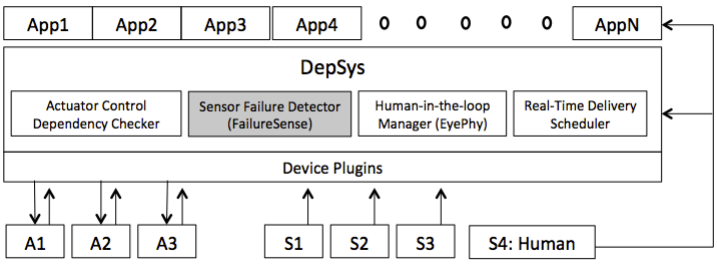


Figure 5.1: FailureSense in DepSys

5.1 Motivation

Recent experience [1] on large scale and long term sensor deployment in residential homes has identified that although the failure rate of a single sensor is low, when we consider hundreds of simultaneously deployed sensors over long periods, that leads to weekly or even daily sensor failure. Table 5.1 shows some sensor failure statistics from some deployments in [1]. The deployments had a wide range of sensors ranging from 47 to 217 deployed for 25 to 44 weeks. From Table 5.1 we see that the average number of sensor failures per day is at least one for each home and for one home it is more than 30. Sensors just don't die, they experience failure in a variety of ways. For example, sensors installed on furniture are moved or covered and produce invalid data. Sensors get dislodged not only due to regular usage, but due to guests, cleaning services and other non-residents. Detecting these types of failures is not the traditional fault detection foci, but will be increasingly important as some predict that smart homes will have hundreds of sensors deployed in the near future.

Motivated by real examples, we develop new schemes to detect not only *fail-stop* failure, but *obstructed-view* and *moved-location* failures that are not addressed well enough to date. *Fail-stop* failure happens when a sensor fails and stops reporting. It may happen due to a hardware failure or power outage. *Obstructed-view* failure happens when a sensor's field of view is compromized. A motion sensor experiences *obstructed-view* failure when it is blocked because some furniture is placed in front of it, or a magnetic sensor may be obstructed because an appliance with a motor is placed near it. A sensor experiences *moved-location* failure if it is moved to a different location. It may happen if the sensor gets dislodged and someone puts it in a different place. It may also happen if it is mounted on furniture and the furniture is moved to a different place.

State of the art techniques [44] [47] that detect *fail-stop* failure cannot detect *moved-location* failure as the sensor is still responding, but the reported values are incorrect. SMART [49] and RFSN [50] detect *non-fail-stop* failure, where a sensor doesn't die completely, but reports incorrect values. But SMART requires non-trivial effort in training. Also, it is not scalable in detecting multiple failures if multiple sensors fail within a short period of time. But as shown in Table 5.1, the potential risk of multiple sensor failures within a short period of time can be high as we observed more than 30 sensors to fail per day on average in one of the homes. Our solution can detect multiple sensor failures even if they fail simultaneously and it requires less training effort than SMART. Another state of the art solution, RFSN requires sensor redundancy to detect sensor failure. Deploying redundant sensors in a home not only increases the cost of deployment, it creates unnecessary channel contention and hurts the aesthetics of the rooms. Our solution doesn't require sensor redundancy, although it is permitted.

In this chapter, we present a novel technique, called FailureSense that detects not only *fail-stop* failure, but *obstructed-view* and *moved-location* failures by monitoring the usage of electrical appliances in the home. FailureSense

House	Deployment Duration (days)	# Sensors Deployed	Sensor Downtime (#days*#sensors)	Avg Sensor Failure per Day
House G	308	217	11346	37
House H	273	67	786	3
House I	224	157	5789	26
House J	175	47	432	3

Table 5.1: Some statistics of sensor failure from four real-home deployments. Computed from [1].

runs as a service in DepSys as a part of the ”Sensor Failure Detector” component as shown in Figure 5.1. The idea behind the proposed solution is, a significant portion of our activities involve handling electrical appliances. When someone turns on an electrical appliance, e.g., light, fan, microwave, stove, dishwasher, washing machine etc., he has to be physically present to turn it on. We realize that this assumption may not hold for all the electrical appliances, e.g., someone can turn on the TV by a remote, but still it holds for a significant portion of electrical appliances in the home and a careful selection of the appliances suffices. Thus, whenever someone turns on the selected appliances, there will be a positive correlation between the appliance activation event and the firing of the motion sensor that is covering that area. Since the movements of the residents are constrained by the floorplan of the house, with sufficient training data, we learn *all the regular intervals* between sensor firing and appliance activation and report a failure when we see a significant deviation from the *regular* behavior.

There are several advantages in using the power infrastructure to monitor electrical appliance usage to detect a sensor failure. First, the power infrastructure does not fail as often as sensors do. Correlation based techniques rely on sensor redundancy and can't detect a failure when the redundant sensors also fail. Relying on the power infrastructure will significantly reduce this dependency. Secondly, some electrical appliances are turned on in a periodic fashion, e.g., we turn on lights in our rooms almost every night. So, we can compute the failure detection latency of sensors based on electrical appliance usage of the residents. Thirdly, with much emphasis on energy saving and smart energy management, power meters may provide real-time appliance specific energy usage in the near future. We already have several COTS devices, e.g., TED [109], and eMonitor [110] in the market and as these technologies become more popular, our solution can get a free ride and detect sensor failure almost free of cost.

5.2 Contributions

This chapter makes three major research contributions. First, we are the first to show that activation events of the electrical appliances in the home can be useful for detecting not only *fail-stop* failure, but importantly *obstructed-view* and *moved-location* failures that are common in smart homes and barely addressed in real deployments or in the literature to date. Second, our solution requires minor training effort, it is scalable in detecting multiple sensor failures even if they fail simultaneously, and it doesn't require sensor redundancy, thus saves cost of redundant sensor deployment, avoids unnecessary channel contention, and improves the aesthetics of the rooms at homes. Third, by

House	# Days	# Motion Sensors	# Electrical Appliances	# Turn on/off Events
A	15	15	10	660
B	35	8	8	1921
C	21	5	1	237

Table 5.2: Summary of data collection from three real-home deployments.

using data from three real home deployments of over 71 days and 2818 recorded turn on and off events of 19 monitored appliances, we observe that our solution can detect *obstructed-view*, *moved-location* and *fail-stop* failures with 82.84%, 90.53%, and 86.87% precision, respectively, with an average of 88.81% recall.

5.3 FailureSense Solution

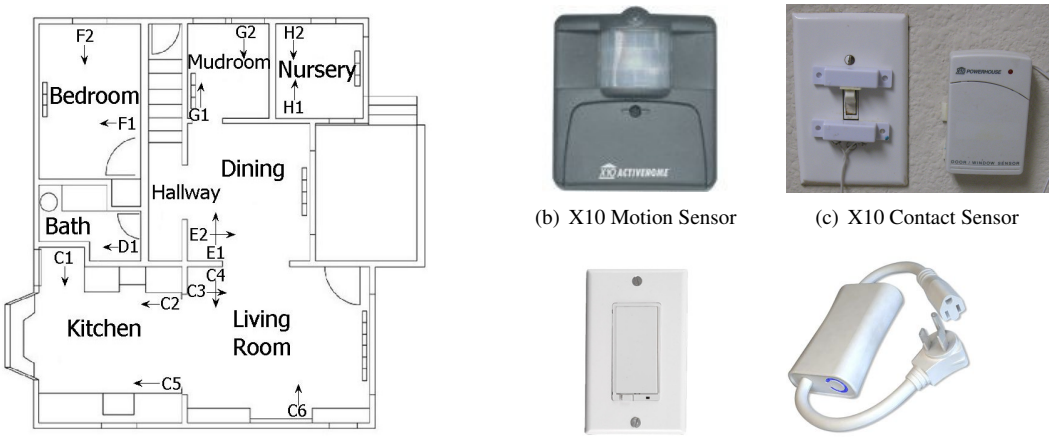
FailureSense has two major steps for detecting a sensor failure. First, based on the training data, it learns and models the *regular* behavior of a sensor with respect to electrical appliance activation at home. Second, it monitors sensor behavior continuously and reports a failure when it observes a significant deviation from the regularity. In this section, we describe our assumptions, data collection procedure, empirical study to model *regular* behavior of sensor firing with respect to appliance usage, and how we use this model to detect a sensor failure.

5.3.1 Assumptions

We assume that sensors do not fail during the training period. We also assume that sensors and electrical appliances are stationary, i.e., they are not moved from their original position in the training period. This assumption does not hold for all the electrical appliances. But as long as it holds for some appliances in each room, like light switches, our solution will work. We also assume that electrical events activation information is available to us. Energy disaggregation is itself an active area of research and some state of the art techniques are mentioned in Chapter 2.5. We are focusing on failure detection instead of detection and classification of electrical appliance activation.

5.3.2 Deployment and Data Collection

To learn how motion sensors fire with respect to turn on/off events of electrical appliances, we use data from two publicly available datasets [1] (Houses A and B) and collect data from one real-home deployment (House C). Overall we use 15 days of data from House A, 35 days of data from House B, and collect 21 days of data from House C. The number of motion sensors, the number of electrical appliances monitored, and the number of turn on and turn off events of the monitored appliances in each house is shown in Table 5.2. Houses A and C are 4-person homes whereas House B is a 3-person home. The floorplan along with the positions of the motion sensors of House A is shown in Figure 5.2(a). The deployment at House C is mainly targeted to detect *obstructed-view* failure. Since we do not have any control over



(a) House A (b) X10 Motion Sensor (c) X10 Contact Sensor

(d) Zwave Light Switch (e) Zwave Plug

Figure 5.2: Floorplan and positions of motion sensors in House A and the sensors that are deployed in houses.

the deployment at Houses A and B, and to detect *obstructed-view* failure, we need to obstruct some sensors' view, we deploy a few sensors at House C and obstruct their views.

The following types of motion sensors and electrical appliances are used in our experiment:

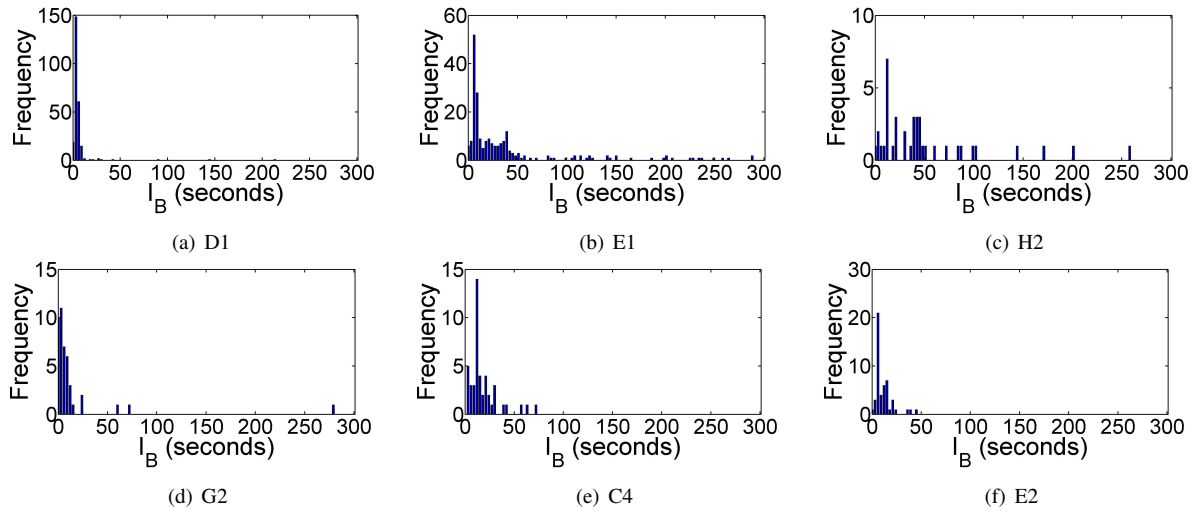
- **Motion Sensors:** X10 motion sensors (Figure 5.2(b)) are used in all homes since the motion sensors manufactured by X10 are inexpensive. We collect the timestamp and sensor ID of each sensor firing.

- **Electrical Appliances:** Several electrical appliances are available in homes. But we use only lights in all homes since almost every room has at least one light and light switches are usually stationary. Since energy disaggregation is not our focus, ZWave Light Switch (Figure 5.2(d)) is used in House A, Zwave Light Switch (Figure 5.2(d)) and Zwave Plug (Figure 5.2(e)) are used in House B, and X10 contact sensor (Figure 5.2(c)) is deployed in House C to figure out when lights are turned on/off. We collect the timestamp and appliance ID of each turn on and turn off events of these electrical appliances.

5.3.3 Empirical Study and Sensor-Appliance Behavior Model

Based on a total of 71 days of collected data from three multi-person homes having 2818 turn on and off events of 19 monitored appliances, we try to understand the sensor firing pattern with respect to the usage of electrical appliances. We monitor the interval between appliance turn on/off events and sensor firing and observe regularity in intervals for some sensor-appliance pairs.

As we mentioned before, for most of the electrical appliances, when they are turned on, someone has to be physically present to turn them on. So, if there is a motion sensor nearby, it should fire *before* the turn on event and *after* the turn on event. If the appliance is turned on the same way all the time, it would take the same amount of time for the sensor to fire before and after the turn on event. However, in reality, people may not access the electrical appliances the same



(a) D1 (b) E1 (c) H2

(d) G2 (e) C4 (f) E2

Figure 5.3: Frequency distribution of I_B of sensors (a) D1, (b) E1, and (c) H2 with respect to the bathroom light and of sensors (d) G2, (e) C4, and (f) E2 with respect to the mudroom light of House A.

way all the time. Since the movements of the residents are constrained by the floorplan of the house, there are only a few ways to reach to the electrical appliances. If we can collect enough training data, we are able to capture different possible ways the residents can reach the electrical appliances. For some electrical appliances we should observe the similar behavior for the turn off events too.

To characterize the sensor firing pattern with respect to appliance usage, we define 3 parameters: I_A, I_B, and $window$. Let I_A $(Interval\ After)$ be the smallest interval between a turn on/off event of an appliance and a sensor firing where the sensor firing happens *after* the turn on/off event. Similarly, let I_B $(Interval\ Before)$ be the smallest interval between a turn on/off event of an appliance and a sensor firing where the sensor firing happens *before* the turn on/off event. Note that I_A and I_B are different for different sensor-appliance pairs. We use a $window$ of 5 minutes and don't consider any sensor firing before/after 5 minutes of the turn on/off events of the appliances. We keep it fixed for all the homes and in all the experiments. Our subsequent analysis shows that I_A and I_B are useful *features* to detect *fail-stop* failure, *obstructed-view* failure, and *moved-location* failure.

As an example to see how these parameters reveal sensor firing patterns with respect to appliance usage, let's consider some sensors and electrical appliances from House A. The floorplan and the position of all the motion sensors of House A are shown in Figure 5.2(a). We choose the bathroom light and the mudroom light as appliances. We discretize I_B values into 3 seconds bins and plot the frequency distribution of I_B in Figure 5.3. Note that only the D1 sensor is in the bathroom, and other sensors are in different rooms when looking at Figures 5.3(a), 5.3(b), and 5.3(c). If D1 suffers from *fail-stop* failure, *obstructed-view* failure or it is moved to some other room, the frequency distribution of I_B of D1 with respect to the bathroom light is going to change. I_A also shows a similar distribution. The bathroom light clearly helps in detecting the failure of D1 sensor. It may also help in detecting failure of other sensors, e.g., E1. For other sensors, we may need to consider other appliances.

Capturing regularity in intervals can be challenging since a sensor-appliance pair can have multiple regular intervals. To capture all the regular intervals of a sensor-appliance pair, based on the empirical study, we use a **Gaussian Mixture Model (GMM)** to represent the distribution of I_A and I_B. A GMM has q components, where each component captures one regular interval. The probability density function of a GMM is a weighted sum of the q component Gaussian densities as shown in the following equation:

$$p(\boldsymbol{x}|\lambda) = \sum_{i=1}^{q} w_i g(\boldsymbol{x}|\boldsymbol{\mu_i}, \boldsymbol{\Sigma_i}), \tag{5.1}$$

where $\boldsymbol{x}$ is a D variate observation, w_i are the mixture weights and $g(\boldsymbol{x}|\boldsymbol{\mu_i}, \boldsymbol{\Sigma_i})$ are the Gaussian densities of the components, where $i = 1, 2, 3, ..., q$. Each component density follows a Gaussian distribution with mean vector $\boldsymbol{\mu_i}$ and covariance matrix $\boldsymbol{\Sigma_i}$ as shown in the following equation:

$$g(\boldsymbol{x}|\boldsymbol{\mu_i}, \boldsymbol{\Sigma_i}) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma_i}|^{1/2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_i})'(\boldsymbol{\Sigma_i})^{-1}(\boldsymbol{x}-\boldsymbol{\mu_i})} \tag{5.2}$$

Mixture weights satisfy the constraint that $\sum_{i=1}^{q} w_i = 1$. The complete GMM is parameterized by the mean vectors, covariance matrices, and mixture weights from all the component densities. These parameters of the GMM are collectively represented by the following notation:

$$\lambda = \{w_i, \boldsymbol{\mu_i}, \boldsymbol{\Sigma_i}\}\, i = 1, 2, 3, ..., q. \tag{5.3}$$

In our case, we have separate Gaussian Mixture Models for I_A and I_B and our $\boldsymbol{x}$ is a single variate observation of I_A or I_B. The parameters of the mixture model (λ) are estimated using the **Expectation Maximization (EM)** algorithm from the training data. When choosing λ, we try using 1-4 components and choose the one that minimizes the Akaike Information Criterion (AIC), which is a measure of relative goodness of fit of a statistical model. Note that it is not necessary for all the sensor-appliance pairs to follow their I_A and I_B according to GMM. We just need enough appliances so that each sensor is covered by at least one appliance, which we find an easy requirement for all the three houses based on the empirical study.

5.3.4 Appliance Selection

For each appliance, we compute the probability of each sensor firing within the *window* of appliance activation events. For each sensor, we choose top k appliances with which the sensor firing is most probable. We *associate* these appliances with these sensors. It means that the failure detection of these sensors only depends on these appliances. The higher the value of k, the lower are the failure detection latency and precision of failure detection. The value of k can

be chosen based on application requirement. We use k = 2 in the evaluation since the instrumented appliances were limited in our data collection.

5.3.5 Online Failure Detection

After modeling the *regular* behavior of sensor firing during the offline training phase, we monitor the sensor-appliance behavior in terms of I_A, I_B online and report a failure when we observe a deviation from the distribution according to some thresholds (described below). More specifically, at runtime, we detect sensor failure using the following steps:

1) Appliance Monitoring: For the selected appliances, we monitor when the appliances are turned on/off.

2) Sensor Monitoring: When a selected appliance is turned on, we monitor the firing of the *associated* sensors within the duration of $window$.

3) Probability Computation: Based on appliance usage and sensor firing, we compute i_A and i_B, which are the observed values of I_A and I_B, respectively. We also compute p_A and p_B, the probabilities of observing these i_A and i_B using equations (5.4) and (5.5) as follows:

The probability of observing $I_A = i_A$ is,

$$p_A = p(I_A = i_A | i_A \leq window) * p(i_A \leq window) \tag{5.4}$$

The reason for having these two separate terms is because we only consider I_A and I_B values within $window$ at the time of modeling GMM. We compute $p(\ i_A \leq window)$ directly from the training data, and $p(I_A = i_A \mid i_A \leq window)$ is computed from the GMM using equation (5.1). Similarly, the probability of observing $I_B = i_B$ is,

$$p_B = p(I_B = i_B | i_B \leq window) * p(i_B \leq window) \tag{5.5}$$

4) Failure Decision: We decide the state of the sensor from the values of p_A and p_B. If $p_A \leq T_A^{low}$ *and* $p_B \leq T_B^{low}$ for N times in a row, we report a failure of the associated sensor, where T_A^{low}, T_B^{low}, and N are thresholds. If $p_A \geq T_A^{high}$ *or* $p_B \geq T_B^{high}$ for M times in a row, we report that the sensor is working fine, where T_A^{high}, T_B^{high}, and M are thresholds. Also, to avoid flooding of reports, after sending one status report, we suppress all the subsequent reports related to that sensor for the next 6 hours.

5.3.6 Threshold Selection

The performance of FailureSense largely depends on the selection of the thresholds. Overall, we use 6 thresholds: T_A^{low}, T_A^{high}, T_B^{low}, T_B^{high}, N, and M at the time of deciding sensor failure status. These thresholds are specific to sensor-appliance behavior and a static threshold may not work for all possible sensor-appliance pairs in all homes. So, we compute these thresholds by taking into account specific sensor-appliance behavior from the training data. Note that the computation is fairly automatic and it doesn't require any user involvement other than turning on/off appliances

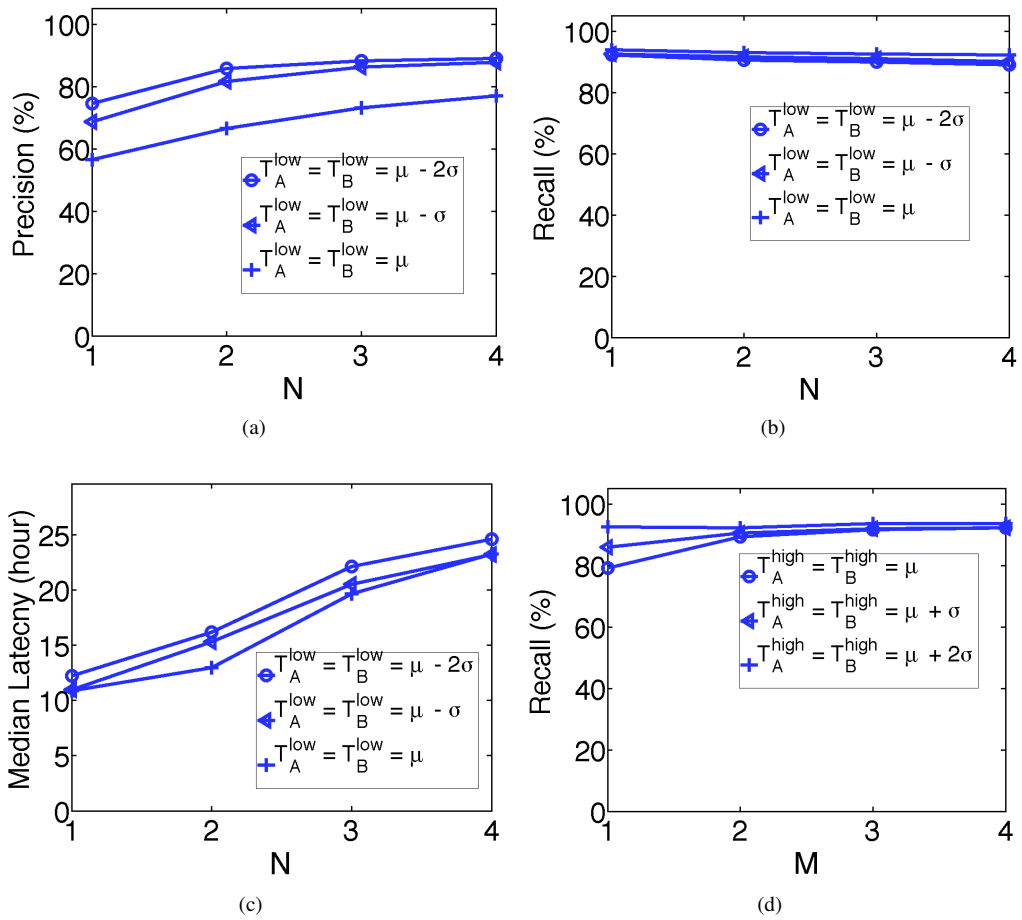


Figure 5.4: Effect of N, T_A^{low}, T_B^{low} on (a) precision, (b) recall, (c) median latency of failure detection and effect of M, T_A^{high}, T_B^{high} on (d) recall in detecting three types of failure in three houses.

while they perform their activities of daily living. The goal is to choose threshold values that maximize precision and recall while minimizing failure detection latency.

Selection of T_A^{low}, T_B^{low}, and N

We report a failure of the associated sensor if we observe $p_A \leq T_A^{low}$ *and* $p_B \leq T_B^{low}$ for N times in a row. So, as N increases, the latency to detect a failure also increases. As N increases, we report fewer number of failures which reduces the recall of detecting a failure, but increases the precision. As T_A^{low} or T_B^{low} increases, the frequency of reporting a failure increases, which in turn increases the recall, but decreases the latency and precision.

We compute mean (μ) and standard deviation (σ) of the p_A, p_B values. We change N from 1 to 4 and change T_A^{low}, T_B^{low} using μ and σ of the p_A, p_B values, respectively, while keeping the other thresholds unchanged and show the impact on average precision, recall, and median latency in Figures 5.4(a), 5.4(b), and 5.4(c), respectively in detecting the three types of failure in three houses (experimental setup along with how precision and recall are computed is specified in the next section). We see that as N increases, precision and latency of failure detection increase, but recall

decreases. Also, as T_A^{low} and T_B^{low} increase, recall increases, but precision and latency decrease, as expected according to our analysis. We choose N = 3 and select T_A^{low}, T_B^{low} to (μ - 2 $*$ σ) of the p_A, p_B values, respectively, which provides high precision at a little loss of recall with a reasonable median latency. Other threshold values can be chosen if that satisfies application requirement.

Selection of T_A^{high}, T_B^{high}, and M

We report the associated sensor is working fine when $p_A \geq T_A^{high}$ *or* $p_B \geq T_B^{high}$ for M times in a row. As these three thresholds T_A^{high}, T_B^{high}, and M are not used in reporting a failure, they don't have any impact on precision and latency of failure detection. But they do have impact on recall. As M increases, we deliver fewer number of reports saying that the sensor is working fine, which increases recall of failure detection. As T_A^{high} or T_B^{high} increases, sensors pass the thresholds fewer number of times, which again increases recall.

We compute mean (μ) and standard deviation (σ) of the p_A, p_B values. We change M from 1 to 4 and change T_A^{high}, T_B^{high} using μ and σ of the p_A, p_B values, respectively, while keeping the other thresholds unchanged and show the impact on average recall in detecting the three types of failure in three houses in Figure 5.4(d) (experimental setup is described in the next section). This figure shows that increasing M increases recall. Increasing T_A^{high}, T_B^{high} also have a similar impact, as expected according to our analysis. We choose M = 2 and select T_A^{high}, T_B^{high} to μ of the p_A, p_B values, respectively for the evaluation. Setting T_A^{high}, T_B^{high} to (μ + 2 $*$ σ) of the p_A, p_B values slightly increases the recall. However, choosing such high threshold values drastically reduces the number of reports saying that the sensor is working fine, which may not be appropriate for some sensors.

This analysis is useful in choosing thresholds to meet application requirements. Note that we do not compute separate thresholds for detecting different types of failure. We compute a single set of thresholds using the above technique and that works in three houses we tested in detecting all the three types of failure. At the time of reporting a sensor failure, we do not specify the type of sensor failure.

5.4 Performance Evaluation

The evaluation is based on data collected from three real homes. The data collection procedure is described in Section 5.3.2. We use 15 days of data from House A, 35 days of data from House B, and collect 21 days of data from House C. The performance of FailureSense is evaluated through a *post-facto* analysis using these datasets, instead of deploying the FailureSense system in another home. Hence, the implementation of FailureSense is a Matlab program that analyzes the collected sensor data and reports a sensor failure using the strategy specified in Section 5.3. It also means that FailureSense is agnostic to the purpose of sensor deployment and the applications that are using the sensor data.

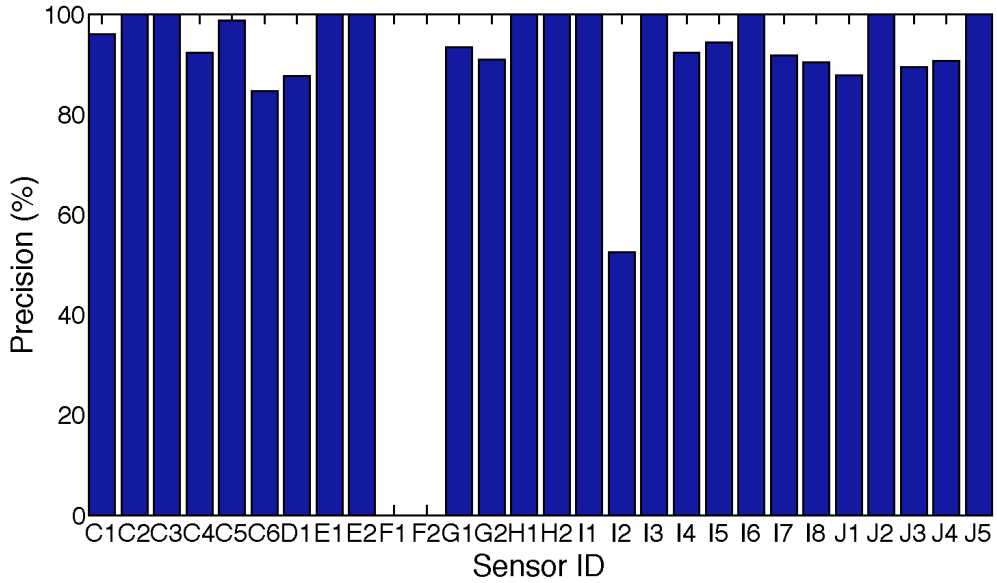


Figure 5.5: Precision of detecting *fail-stop* failure at House A (sensors C1 - H2), House B (sensors I1 - I8), and House C (sensors J1 - J5).

We evaluate the performance of our algorithm in terms of precision (% of failure reports where the sensor is actually failed) and recall (% of failed sensor reported). The way we compute these metrics are, when FailureSense reports a sensor status, we compare with the ground truth state of the sensor and determine whether this report is a true positive (TP, i.e., we report a failure when it has failed), false positive (FP, i.e., we report a failure when it has not failed), true negative (TN, i.e., we report a sensor has not failed when it has not failed), or false negative (FN) (i.e., we report a sensor has not failed when it has failed). Then we compute precision = TP/(TP + FP) and recall = TP/(TP + FN). Failure detection latency largely depends on how frequently the residents use the appliances. We discuss the frequency of appliance usage and its implication in failure detection latency in Section 5.6. The experimental setup is different for different types of failure.

5.4.1 *Fail-stop* failure

We simulate the behavior of *fail-stop* failure by discarding all the readings of the failed sensor after it fails. We evaluate it on all the sensors of all the three houses containing 28 sensors. We perform 3 fold cross validation for the evaluation. More specifically, for Houses A and C, we train on 10 days of data and test on 5 days of data. For House B, we train on 24 days of data and test on 11 days of data. The reason for using only 15 days of data from house C is because, the sensors' views were obstructed at the 16th day to evaluate *obstructed-view* failure (c.f. Section 5.4.2). The sensor failure day is chosen to be the first day of the testing period when computing true positive and false negative, and the last day of the testing period when computing false positive and true negative. Whether it is the first or the last day, we randomly choose 100 timestamps within that day, fail the sensor at these timestamps, and show the average results.

The precision of detecting *fail-stop* failure is shown in Figure 5.5. The average precision is 86.87%. The recall is 100% for all the sensors except sensors F1 and F2, for which it is 0. The average recall is 92.86%. We see that our solution detects failure of all the sensors except F1 and F2 of House A. The reason why the solution fails to detect the failure of sensors F1 and F2 is because F1 and F2 are in the bedroom of House A and the dataset of House A doesn't contain turn on/off events of any electrical appliances in the bedroom. That's why sensors F1 and F2 are not associated with any electrical appliances and we can not detect the failures of F1 and F2. Also, the precision is really low for sensor I2 of House B. I2 is deployed in one bedroom of House B and this is the only room where instead of using light switches, a lamp is instrumented with a Zwave plug (Figure 5.2(e)). We are not sure if this is the reason or if this is due to the behavior of the residents, but there were no regularity in intervals for I2 with respect to the lamp. That's why although our solution can detect when the sensor fails, it suffers from a low precision. However, we see that the solution performs well for most of the sensors in three houses.

5.4.2 *Obstructed-view* failure

We evaluate the performance of detecting *obstructed-view* failure by simulating the sensor obstruction in Houses A and B, and by physically obstructing the views of sensors in House C. It is different from the *fail-stop* failure detection in that the failure takes place for a transient period. In House B, we train our solution with 15 days of data and test on the next 20 days of data. Within these 20 days, for each sensor, we randomly choose a 10 day period for which we assume that the sensor view is obstructed and we discard the sensor readings of these 10 days. Since the obstruction can happen at any time, we randomly choose this 10 day period 100 times for each sensor and show the average results. Similarly, for House A, we train our solution with 10 days of data and test on the next 5 days of data. Within these 5 days, for each sensor, we randomly choose a 3 day period of obstruction when we discard the sensor readings. We perform it 100 times and show the average results. At House C, after 15 days of data collection, we obstruct the views of the 5 motion sensors. This is a controlled experiment and we consider arbitrary positions for placing sensors in the home since X10 motion sensors are *do-it-yourself* sensors that end users just buy and place the sensors in their homes without any professional expertise. The way we obstruct them is shown in Table 5.3. We train with the first 10 days of data, evaluate false positive and true negative on the next 5 days of data, we fail the sensors on the 16th day of data collection at 12:00 PM, and evaluate true positive and false negative on the next 5 days of data.

The precision of detecting *obstructed-view* failure is shown in Figure 5.7. The average precision is 82.84%. The recall is 100% for all the sensors except F1, F2, I2, J3, C2, and D1. For sensors F1, F2, I2, and J3, the recall is 0. For sensors C2 and D1, the recalls are 80% and 85.71%, respectively. The average recall is 84.49%. We see that we can not detect the failure of sensors F1, F2, I2, and J3. The reasons for F1, F2, and I2 are, as described in Section 5.4.1, F1 and F2 are deployed in the bedroom of House A and the dataset does not contain any turn on/off events of any electrical

Sensor ID	House ID	How sensor view is obstructed
J1	C	It is mounted at the wall of the living room. A couch is moved that blocks its view.
J2	C	It is mounted at the wall of the dining room. Some moving boxes are placed in front of it.
J3	C	It is mounted at the wall of the kitchen (Figure 5.6(a)). A cabinet is opened that blocks its view (Figure 5.6(b)).
J4	C	It is mounted at one cabinet of the kitchen. Its view is blocked when the cabinet is opened.
J5	C	It is mounted at the refrigerator. We assume that it gets dislodged and someone puts it at the top of the refrigerator.

Table 5.3: Experimental setup for *obstructed-view* failure detection.

(a) (b)

Figure 5.6: Experimental setup for *obstructed-view* failure. (a) Sensor J3 before obstruction. (b) Sensor J3 after obstruction.

appliances of that room, and sensor I2 is deployed in one bedroom of House B and there were no regularity in intervals for I2 with respect to the lamp instrumented with a Zwave plug (Figure 5.2(e)) in that room. The reason for not being able to detect the failure of sensor J3 is, although the view of J3 is obstructed and it can not see any motion, it used to fire every time a light switch, which is close to J3, is turned on/off. It may be because the light switch is instrumented with a contact sensor (K1 in Figure 5.6(a)) and turning the light switch on/off causes a change in the magnetic field of the contact sensor and causes a small light in front of the contact sensor to blink. We are not sure exactly what caused J3 to fire every time the light switch is turned on/off. But it shows a potential caveat of applying the solution. Appliance selection can be tricky and if the appliance is close, it may cause disturbances.

5.4.3 *Moved-location* failure

We simulate the behavior of *moved-location* failure as in [49] by replacing the failed sensor's data with the data produced by the sensor at its new position. We evaluate it in Houses A and B since the deployment in these houses spans the

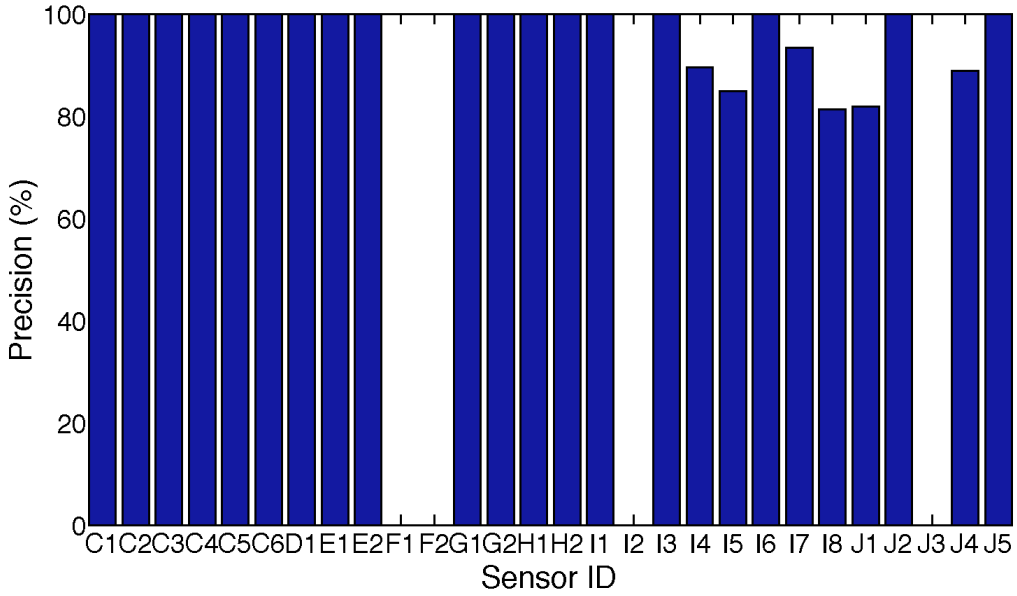


Figure 5.7: Precision of detecting *obstructed-view* failure at House A (sensors C1 - H2), House B (sensors I1 - I8), and House C (sensors J1 - J5).

whole house. We do not use House C in this evaluation as it has a small scale deployment mainly targeted to evaluate *obstructed-view* failure detection. We perform 3 fold cross validation for the evaluation, as described in *fail-stop* failure detection.

The precision of detecting *moved-location* failure is shown in Figure 5.8. The average precision is 90.53%. The recall is 95% or more in all these cases, except in E2 $\rightarrow$ H1, C2 $\rightarrow$ C6, H1 $\rightarrow$ H2, E1 $\rightarrow$ D1, and I8 $\rightarrow$ I3, where the recalls are 91.83%, 55.30%, 0, 85.71%, and 83.71% , respectively. The average recall is 89.09%. We select a pair of sensors and move the first one to the position of the second one, e.g., in the first case of Figure 5.8, sensor G1 is moved to H1's location. The positions of sensors along with the floorplan of House A are shown in Figure 5.2. The dataset doesn't offer us the floorplan of House B. However, from the description, we know that all the sensors of House B in Figure 5.8 are in different rooms. Our solution works well if sensors are moved from one room to another room. However, if a sensor is moved slightly within the same area, it doesn't cause a significant deviation from the distribution of I_A and I_B and the solution doesn't perform well. This is why when C2 is moved to C6 and H1 is moved to H2, the recalls are low. Note that small sensor displacement may not make any difference in application semantics. On these cases, this type of minor movement is not a movement failure and is not detected as such.

5.5 Comparison with state of the art

State of the art solutions that detect failure relevant to this work either use correlation based technique (RFSN [50]) or use a classifier based approach (SMART [49]). We compare our solution with both types of strategies in terms of sensor

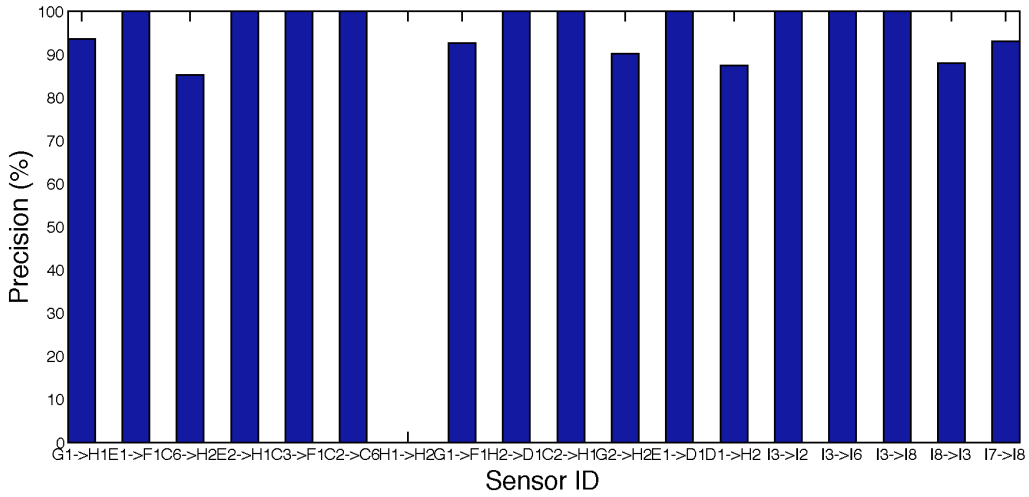


Figure 5.8: Precision of detecting *moved-location* failure at House A (sensors C1 - H2) and House B (sensors I1 - I8).

redundancy, training effort, and scalability in detecting multiple sensor failure at the same time.

5.5.1 Comparison with RFSN

Reputation-based Framework for Sensor Networks (RFSN) [50] detects misbehavioring sensors by allowing each sensor to maintain reputation of the neighboring ones. It doesn't work when there is no sensor redundancy as each sensor needs to monitor the correlation with the neighboring sensors to detect sensor failure. Even if there is sensor redundancy, if the neighboring sensors are compromised or failed, then building of the reputation metrics may fall into jeopardy. To detect the failure of a sensor, our solution doesn't rely on any neighboring sensors and it works in presence of no redundant sensors. Instead, it relies on power infrastructure, which is less likely to fail than the neighboring nodes.

RFSN is suitable where sensors are located near each other and deliver real-valued data at a constant rate, e.g., a temperature monitoring system where each sensor reports its temperature reading in every minute. However, our solution is suitable for event-driven applications where sensor values are usually binary. To adapt RFSN for event-driven applications with binary valued data, we make the following changes in RFSN as suggested in [49]. First, since sensor values are binary, we use temporal correlation instead of value based correlation, i.e., sensors have higher correlation if they fire together within a short duration. Second, since sensors are event-driven and most of the time the sensors are idle, to achieve meaningful temporal correlation we ignore the periods when no sensors fire.

We evaluate the performance of RFSN and FailureSense when multiple sensors experience *fail-stop* failure at House A and show it in Figure 5.9. There are 15 motion sensors deployed at House A. We vary the number of failed sensors from 1 to 15. For i number of sensor failures, we randomly select i sensors, run the experiment 100 times and show the average % of sensor failure detected by both algorithms in Figure 5.9. As shown in Figure 5.5, FailureSense fails to detect the failure of F1 and F2 sensors, whereas RFSN fails to detect a failure when all the redundant sensors also fail.

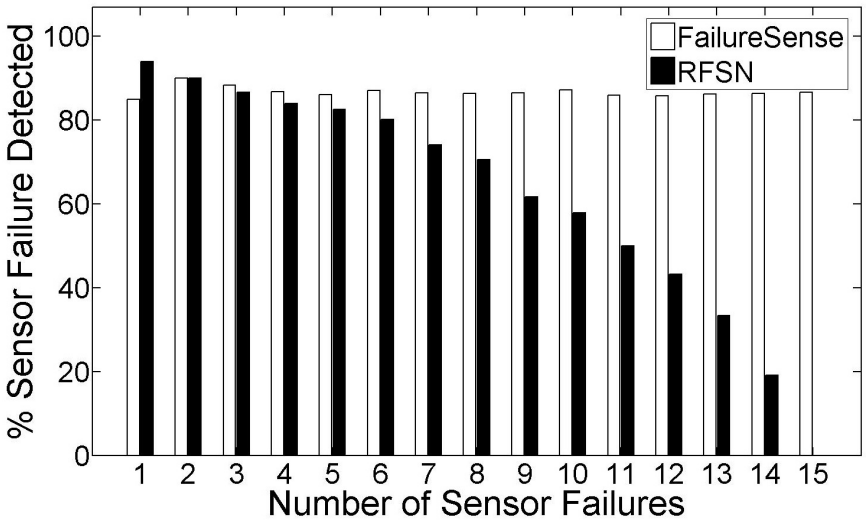


Figure 5.9: Percentage of sensor failure detected by FailureSense and RFSN when multiple sensor experiences *fail-stop* failure at House A.

It shows that FailureSense maintains an average of 86.69% sensor failure detection across all *fail-stop* sensor failures where RFSN performance degrades when more sensors fail.

5.5.2 Comparison with SMART

SMART [49] detects *non-fail-stop* failure by analyzing the relative behavior of multiple classifier instances trained to recognize the same set of activities based on different subset of sensors. It requires *activity labeling* for each house to understand how each sensor relate to each activity, which is a non-trivial amount of work for the training. We do not need activity labeling and our solution reduces the training effort significantly (c.f. Section 5.6.1). Also, SMART requires training of the classifier instances for all possible combinations of sensor failure as because different combinations of sensor failure affect classifier instances differently. If there are n sensors in a house, their approach requires $2^n - 1$ combinations of node failure analysis, which is not scalable. The authors admitted this limitation and evaluated their solution assuming single node failure. Our solution overcomes this limitation as we do not need different training to detect different combinations of sensor failure. We could not make a direct comparison with SMART due to the unavailability of datasets. Because, the dataset we have doesn't have activity labeled and the dataset that SMART uses doesn't have the turn on/off events of electric appliances. But admittedly SMART is not scalable in detecting multiple sensor failure at the same time and the scalability of our solution is shown above.

5.6 Discussion

In this section, we discuss several aspects of our solution including the training effort, latency of failure detection, availability of smart energy meters, dependency on human behavior, and generalization of the solution to detect failure

of other types of sensors.

5.6.1 Training Effort

We need training effort to accomplish two tasks:

1. **Appliance Activation Detection:** We need to know which appliances are turned on/off from the power infrastructure. The training effort largely depends on the selection of appliance activation detection technique. For example, Electrisense [53] can automatically detect and classify the use of electronic appliances at home from a single point of sensing at the power wire using EMI signature. It requires turning on/off five to six times per appliance and some processing time to capture its EMI signature, which is very little.

2. **Learning Regular Appliance-Sensor Behavior:** To learn regular appliance-sensor behavior and build our model, we need to collect 30-40 turn on/off events per appliance. At this phase, the end-user just performs activities of daily living and our system captures timestamps of appliance turn on/off events and sensor firings. The duration of this phase may vary depending on appliance selection and personal preference of using the appliances, e.g., some people may turn on lights several times a day and some people may turn on once and keep them on for the whole day. As an example, we observe that the average and standard deviation of the frequency of turn on and turn off events of bathroom light and mudroom light in House A are 16.20 ± 3.80 and 3.13 ± 3.80 per day. The reason for high frequency for bathroom light is because this is a 4 person home. Based on the empirical study on three houses, we conclude that it may take about 10-15 days of data to build accurate models.

5.6.2 Failure Detection Latency

The failure detection latency depends on appliance selection, the frequency of appliance usage, and the selection of threshold values. As an example, for the experimental setup in Section 5.4, the average median latency of detecting *fail-stop*, *obstructed-view*, and *moved-location* failures are 20.03 hours, 17.69 hours, and 28.52 hours, respectively. The average median latency for detecting these three types of failure is 22.08 hours, which is reasonable for many applications, including remote health monitoring and energy management systems. However, for emergency health care and security services, this latency may be inadequate. Usually people purchase more expensive solutions for these applications.

5.6.3 Smart Energy Meter Requirement

Although our solution doesn't require sensor redundancy, it requires a smart energy meter that can detect appliance activation events. This requirement may look like another type of redundancy as such meters are not widespread yet.

However, based on a recent study [111], the worldwide installed smart electricity meters will grow at a compound annual growth rate of 26.6 percent between 2010 and 2016 to reach 602.7 million. Another survey suggests that people are willing to spend an average of 150$ on energy equipments if they could save as much as 30% of their energy bill [112]. As people are getting more and more conscious about energy usage and monitoring, smart energy meters may become an indispensable part of future smart homes.

5.6.4 Results with Actual Smart Energy Meter

Our results assume that smart energy meters can detect appliance activation events with 100% accuracy. In actual practice, although the state of the art techniques are not 100% accurate, their accuracy is considerably high, e.g., 93.82% of ElectriSense [53] and 90% of [54]. Our solution also assumes that smart meters are more reliable than the deployed sensors, which we found to be true based on deployment experience with the eMonitor [110] energy management and X10 motion and contact sensors.

5.6.5 Dependency on Human Behavior

The performance of failure detection relies on the behavior of the residents. If someone turns on all the lights and never turns them off, then we will not be able to detect sensor failures. However, based on data collected from three real homes, we see that people actually turn on and turn off lights almost everyday and thus it is reasonable to assume such behavior. However, there may be exceptional cases. For example, if there is a party, then appliances may be used in a different way. To handle such exceptional cases, we plan to use Exception Flagging [113], where our solution just ignores the data collected during such period. If the behavior of the residents change after the training period, then our performance may degrade. Putting humans into the loop will allow the system to use human feedback to distinguish between behavior change and sensor failure. When the system gets confused about the failure status of a sensor, it can ask the residents to turn on the associated appliance. It is easier for the end users to turn on appliances than assessing the failure status of a sensor by themselves. We consider it our future work. Note that some state of the art solutions, e.g., SMART [49] also suffer from change of human behavior. Putting humans into the loop may not help all the time. For example, if the sensors fail while the residents are traveling, then our solution will not be able to detect such a failure. However, the end goal of a number of home health care and energy management systems is to monitor activities of daily living [8] [9] [114] [115] [116] and occupancy patterns [64] [65], where system performance may not be affected much due to sensor failure when the residents are away. When they return and start using appliances, sensor failures will be detected.

5.6.6 Generalization to Other Types of Sensors

We believe that our solution is generalizable to various types of sensors, including but not limited to the following types:

1. **Light Sensor:** When a light switch is turned on, the nearby light sensor should see a sudden increase of light intensity within a regular period. If the light sensor is failed, or covered, or moved to some other room, it will not be able to see that change. We can use this property to decide if the light sensor has failed.

2. **Acoustic Sensor:** When someone turns on TV, radio, washing machine, coffee maker, electric shaver, food processor, hair dryer, or a microwave oven, the nearby acoustic sensor should see an increase of sound intensity or even a sound pattern within a regular period. We can use this property to assess the failure state of the neighboring acoustic sensors.

3. **Water Fixture Sensor:** Sensors that are attached to various water fixtures, e.g., a contact sensor attached to a faucet in the bathroom/kitchen, a sensor attached to a bathroom flush, and a sensor for detecting shower do not fire when the room light is turned on. However, they do fire within a short time when water starts to flow through the water fixture. We can use this property for assessing the reliability of the water fixture sensors as we can detect which water fixture is drawing water by using smart water meter and motion sensors as in WaterSense [117].

However, our solution may not be adequate detecting failure of some other types of sensors, e.g., temperature, humidity, and radio activity. But, a lot of home health care and energy management systems use motion sensors, acoustic sensors, and water fixture sensors to monitor activities of daily living [8] [9] [114] [115] [116] and occupancy patterns [64] [65] of the residents. Such systems will greatly benefit from our solution.

5.7 Summary

This chapter presents how DepSys addresses sensor reliability dependencies by employing a novel sensor failure detection scheme called FailureSense. FailureSense addresses not only *fail-stop* failures, but importantly, *obstructed-view* and *moved-location* failures; new types of failures common in smart homes that are very difficult to detect even if sensors are made highly reliable and barely addressed to date in the literature or in real deployments. Our solution is the first one to detect these failures without requiring sensor redundancy and with minimal training effort. The performance evaluated in three homes exhibits that FailureSense can detect *obstructed-view*, *moved-location* and *fail-stop* failures with 82.84%, 90.53%, and 86.87% precision, respectively, with an average of 88.81% recall. FailureSense is applicable to most of the common types of sensors found in real deployments. It takes into account complicated and realistic sensor failures in a home setting. Therefore, by using FailureSense, DepSys addresses apps' dependencies on sensor reliability status significantly.

Chapter 6

Addressing Human-in-the-loop Dependencies

Humans are intimately involved with the integrated smart home systems. Some smart home apps rely on the behavior and the physiological status of the residents for taking appropriate control decision, which raises human-in-the-loop dependency issues across the human-in-the-loop apps. In this chapter, we describe how DepSys's Human-in-the-Loop Manager addresses app dependencies due to human-in-the-loop by employing a novel technique called EyePhy (shown in Figure 6.1). EyePhy also works for addressing the human-in-the-loop dependencies of the smart phone apps.

The rest of the chapter is organized as follows. We discuss the motivation of using EyePhy in Section 6.1. We list the technical contributions that this chapter makes in Section 6.2. Then we describe the EyePhy solution, which includes our proposed architecture (Section 6.3), classification of parameters and dependencies (Section 6.4), stakeholders and their responsibilities (Section 6.5), suggested app metadata (Section 6.6), and installation time and runtime dependency detection and resolution techniques of EyePhy (Section 6.7). We discuss some aspects of the solution in Section 6.9 and provide a summary of the chapter in Section 6.10.

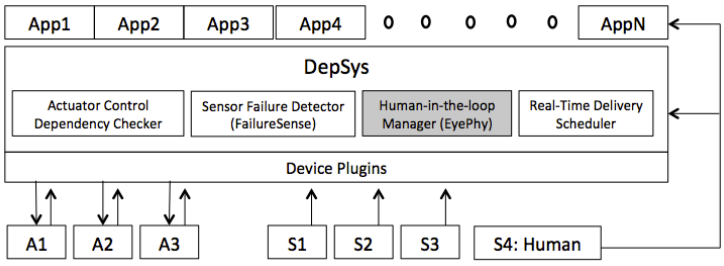


Figure 6.1: EyePhy in DepSys

6.1 Motivation

The US Food and Drug Administration expects that there will be 500 million smart phone users downloading healthcare related apps by 2015 [118]. Currently, Apple and Google each have more than 1 million apps in their app stores. Among these apps, a significant number of medical apps already exist and newer apps are being developed everyday. Many of these apps are human-in-the-loop CPS apps and in addition to sensing, communication, and computation, some apps perform interventions to control some physiological parameters of the human body.

This control part of the human-in-the-loop CPS apps poses several challenges because of two reasons. First, each app is developed independently and when it performs a control action, i.e., intervention on the human body, it does so without any knowledge about how the other apps work. Second, humans are an integral part of the feedback control loop. Therefore, when an app performs an intervention to control its target parameters, it may affect other physiological parameters without even knowing it since a human body has many interconnected parts. As a result, if the user installs redundant apps, multiple apps may administer the same drug independently that may result in a drug overdose. Also, multiple interventions can increase or decrease each other effects, some of which can be harmful to the health of the user. No existing app engines available in the market perform any analysis across apps' control actions on the human body.

Addressing human-in-the-loop dependencies requires a very different technique than that are described in the previous chapters for addressing other types of dependencies. For example, we ask app developers to specify the *effects* of their actuations to address actuator control dependencies (Chapter 3.7). Such a strategy works well in a home setting where there are only a few environmental parameters that can be affected by an intervention. App developers may even skip some effects in a home setting, e.g., an app that increases the room temperature by turning on an HVAC system may increase the humidity a little bit, but this effect is minor and can be ignored and/or is not safety critical. However, we can not ignore such effects in the context of a human body. Such a strategy of specifying the effect of intervention does not work because a human body has many interconnected parts. So, app developers would have to specify hundreds of parameters in order to specify the effects of an intervention on the human body. For example, using a human physiological simulator [73], we see that a single exercise intervention affects about 1500 variables of the human body.

In this chapter, we propose the design of a service, EyePhy that detects dependencies across interventions of human-in-the-loop CPS apps designed for both smart homes and smart phones. We call it EyePhy as it has a closer *eye* on the *phy*siological parameters of the involved human being. EyePhy runs as a service in DepSys as a part of the component named "Human-in-the-loop Manager" as shown in Figure 6.1. EyePhy uses a physiological simulator called *HumMod* [73] that can model the complex interactions of the human physiology using over 7800 variables capturing

cardiovascular, respiratory, renal, neural, endocrine, skeletal muscle, and metabolic physiology. HumMod is constructed by the medical community from the empirical data collected from peer-reviewed physiological literature. Using a physiological simulator to detect dependencies has many advantages. First, it allows us to perform dependency analysis across a wide range of physiological parameters as HumMod models human physiology using over 7800 variables. Second, it enables us to take into account drug dosage and the time gap between the interventions in the dependency analysis. There are websites [119] [120] that can be used to check for potential drug interactions, but they do not take into account these issues. Also, they can't provide interaction analysis between a drug and a non-drug intervention, e.g., exercise that our solution can offer. Third, our dependency analysis can be personalized, e.g., if someone has heart related problems, he can focus the dependency analysis to the heart. And fourth, all of these can be done without much effort from the app developers in specifying dependency metadata.

6.2 Contributions

This chapter makes three major research contributions. First, EyePhy provides the most comprehensive dependency analysis across human-in-the-loop apps to date by modeling the environment, i.e., the human body using over 7800 variables and using the model as a simulator to understand the potential dependencies among interventions. Second, our solution takes into account drug dosage and time gaps between the interventions at the dependency analysis, it requires minimal efforts of the app developers in specifying dependency metadata, and it offers personalized dependency analysis for the user. Third, by using the HumMod simulator, we demonstrate the magnitude of dependencies that arise during multiple interventions in a human body and the significant ability of detecting these dependencies using EyePhy.

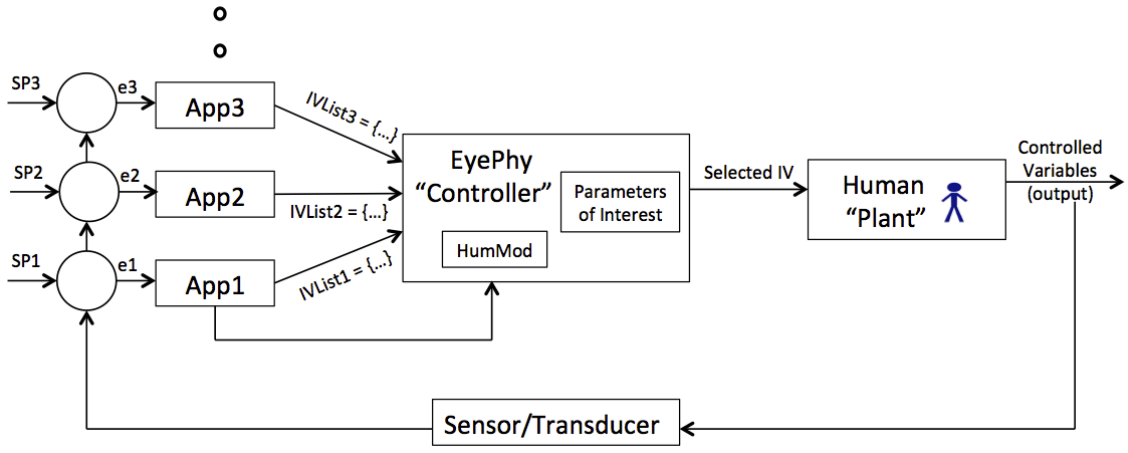


Figure 6.2: Runtime Dependency Detection by EyePhy

6.3 Architecture

We assume a centralized architecture for the app-based paradigm as in DepSys (Chapter 3.2). Such a centralized architecture is also common for smart phones [16] [17]. We assume that apps are run at the top layer by an app engine, which is the OS of a smart phone or a smart home. EyePhy runs as a service for the app engine, and detects dependencies at app installation time and at runtime.

The way EyePhy controls apps' interventions to a human at runtime is shown in Figure 6.2. Each app has one or more physiological parameters of interest that it monitors, if needed, and tries to control. The app may have a setpoint that is used to compute an error, e.g., App1 has setpoint *SP1* and it computes an error *e1* based on that. Then each app offers a list of suggested interventions to EyePhy, e.g., *IVList1* is a list of one or more interventions suggested by App1. The list may contain the same drug with different dosage level, or different drugs, or other interventions. EyePhy uses HumMod to determine potential interactions with other previous interventions and selects an intervention that doesn't conflict with the previous ones. The selected intervention affects the physiological parameters, some of which is captured by the sensor/transducer and being used for determining the next interventions. An app uses its sensor data, if available, to update the values of physiological parameters of HumMod as shown in Figure 6.2 by App1. Figure 6.2 shows how humans are part of the control loop. There are three novelties in our proposed system in Figure 6.2 compared to other human-in-the-loop feedback control systems. First, each app suggests a list of interventions instead of just one intervention, when possible. Second, EyePhy detects interactions among the interventions automatically by using a physiological simulator. Third, it considers the parameters of interest of the user at the time of detecting interactions, which allows dependency analysis to be personalized.

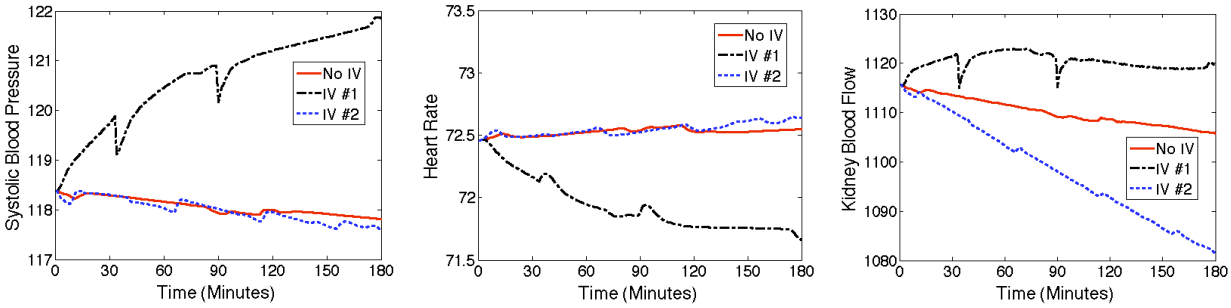
6.4 Parameter and Dependency Classification

HumMod models human physiology using over 7800 variables. If EyePhy uses all the variables for dependency detection, then almost any two interventions will interact with each other. We realize that all interactions are not necessarily harmful to the human body and hence to understand the dependencies, we categorize the parameters and dependencies as follows.

6.4.1 High level and low level parameters

High level parameters are the physiological parameters that are used to assess the general health of a person in most medical settings. These are also called *vital signs*. These parameters are as follows.

1. Body temperature



(a) Systolic Blood Pressure (b) Heart Rate (c) Kidney Blood Flow

Figure 6.3: The effects of two interventions on the (a) Systolic Blood Pressure, (b) Heart Rate, and (c) Kidney Blood Flow.

2. Heart rate

3. Blood pressure

4. Respiratory rate

5. Glucose

However, this list is extensible. We know the ranges (high, low) of each of these variables for a healthy body, which can be a function of age, sex, etc.

Low level parameters are the other physiological parameters that are represented by over 7800 variables in HumMod. Unfortunately, the normal ranges of these parameter values of a healthy person are typically unknown.

An example of how two different interventions can affect the high level and low level physiological parameters is shown in Figure 6.3. We consider two interventions, *InterVention #1* (*IV #1*) and *InterVention #2* (*IV #2*) in this analysis. We also consider the case without any intervention (shown as *No IV* in the figure). We use HumMod to simulate the effect of these interventions on the physiological parameters of a human body. The simulated person takes breakfast from 7 AM to 8 AM in the morning. In case of *IV #1*, we administer a drug called *Digoxin* at 8 AM after eating the meal, which is used to treat heart failure and monitor the physiological parameters for the next 3 hours. In case of *IV #2*, we administer a drug called *Spironolactone*, which is used to treat patients with hyperaldosteronism, at 8 AM after eating the meal independently from the previous run and monitor the physiological parameters for the next 3 hours. In case of *No IV*, we just monitor the physiological parameters for 3 hours from 8 AM without performing any intervention.

We show the effect of these interventions on three physiological parameters (systolic blood pressure, heart rate, and kidney blood flow) in Figure 6.3 for the next 3 hours from the 8AM of the simulated time. When we analyze the effect on the high level parameters (blood pressure and heart rate) we do not see any conflicts between the two interventions. Because, we see in the figure that *IV #1* increases the blood pressure and decreases the heart rate, while *IV #2* has minor effect on these parameters. Hence, they aren't considered conflicting. However, when we observe the effect of these

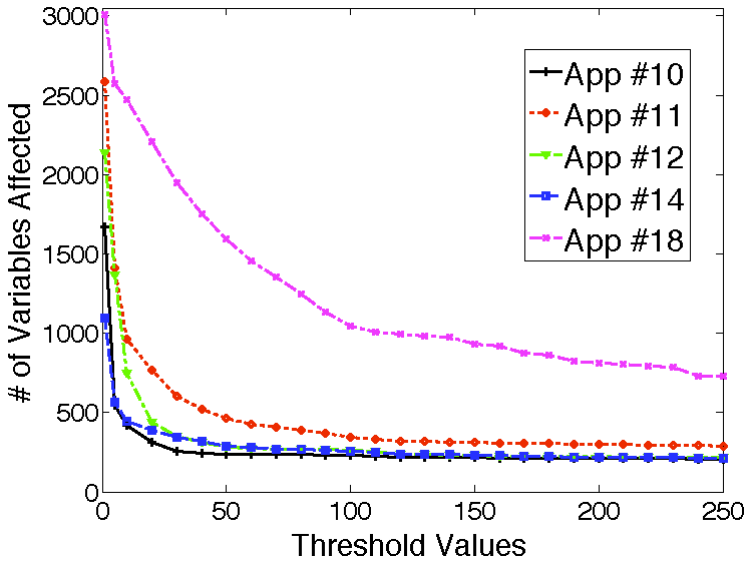


Figure 6.4: Number of variables affected by different apps' interventions with different threshold values

two interventions on the kidney blood flow, which is a low level parameter, we see that they have completely opposite effect there. *IV #1* increases the kidney blood flow, while *IV #2* decreases it. This figure shows that using only high level parameters is not sufficient to detect conflicting interventions.

6.4.2 Developer specified, caregiver specified, and unspecified parameters

We run an experiment to see the how many physiological parameters can be affected by an intervention and check whether it is feasible to ask the app developers to specify all the physiological parameters that can be affected by their interventions. We use interventions performed by App #10, #11, #12, #14, and #18 from Table 6.1 in this study. The version of HumMod that we are using has 7834 variables capturing many physiological parameters. HumMod does not provide any built in technique to determine whether a variable is affected by an intervention. Hence, we use a threshold (X) for each parameter to determine that. More specifically, for each intervention, we run the experiment two times, one with normal behavior (no intervention) and one with the intervention. We run both experiments for 3 hours of simulated time on the human body and collect all the variables' values once per minute. Assume that at minute $t1$, for variable $v1$, we observe $v1$=$n1$ for normal behavior and $v1$=$iv1$ for the intervention. Variable $v1$ is considered affected by the intervention if $|n1 - iv1|/n1 >$ X for at least one observation within the 3 hours. We count the number of variables affected by each intervention for different values of X ranging from 1% to 250% and show it in Figure 6.4. For the interventions that administer drugs (App #10, #11, #12, #14), we administer the app's first preferred dosage as mentioned in Table 6.1. For app #18, we assume that the app suggests to do biking for 30 minutes.

We see in Figure 6.4 that when X is as low as 1%, 1670, 2588, 2137, 1098, and 3010 variables are affected by App #10, #11, #12, #14, and #18, respectively. When X is 5%, 545, 1412, 1370, 564, and 2571 variables are affected by App

#10, #11, #12, #14, and #18, respectively. When X is 250%, 208, 289, 219, 211, and 727 variables are affected by App #10, #11, #12, #14, and #18, respectively. This demonstrates that many physiological parameters are affected by each intervention even if we use high thresholds to choose the affected variables. It clearly shows that it is impractical to ask the app developers to specify all the physiological parameters that can possibly be affected by the app.

App developers mainly specify the high level physiological parameters that can be affected by their interventions. They may also specify low level physiological parameters if known. *Developer specified* parameters are the physiological parameters suggested by the app developers for dependency analysis. *Caregiver specified* parameters are the physiological parameters that are not directly specified by the app developers, but suggested by the user, or by the caregiver, or by the doctor, and considered for dependency analysis by EyePhy. These parameters provide an opportunity to personalize the dependency analysis. *Unspecified* parameters are the parameters identified by EyePhy automatically because of the significant impact on these parameters by the apps' interventions. The unspecified parameters are presented to the users, caregivers, and doctors for additional consideration. If they select any of these parameters, the selected unspecified parameters are treated as caregiver specified parameters in the subsequent analysis.

6.4.3 Primary and secondary dependencies

An intervention can have primary and secondary dependencies. For example, if an app performs an intervention to increase the heart rate, that may increase the blood pressure. Here, increasing the heart rate is representing a primary dependency and increasing the blood pressure is representing a secondary dependency. App developers specify the primary and secondary dependencies of the developer specified parameters.

6.5 Stakeholders and Their Responsibilities

We classify the stakeholders and assign the responsibilities among them as follows.

- App developers: They develop apps and describe the primary and secondary dependencies using the high level parameters (just five parameters, unless extended) as app metadata. They can also use low level parameters if known. In order to ensure all the app developers use the same terminology, we suggest to use the parameter names specified in HumMod.

- Users: They install apps and the installed apps control their physiological parameters. If they have some basic knowledge on the human physiology, they can choose parameters of their own interest, e.g., if someone has heart related problems, he can choose heart related low level parameters. However, we do realize that many users are not knowledgeable enough to choose parameters by their own. In that case, the caregivers and doctors choose the low level parameters of interest as described below.

- Caregivers and doctors: The caregivers and doctors are suggested to maintain a database of low level parameters based on the medical condition of the users. EyePhy pulls the information from the database and performs personalized dependency analysis without requiring the users to know about any low level parameters.
- Physiological model developers: They develop physiological simulator HumMod that can simulate the effect of various interventions on the human body.
- EyePhy developers: We, the EyePhy developers, integrate the efforts of all other stakeholders to provide personalized dependency analysis for the users involving human-in-the-loop.

6.6 App Metadata

EyePhy expects app developers to specify the following meta data.

- The primary and secondary dependencies involving the high level parameters for the interventions performed by the app. They can also specify dependency information of the low level parameters if needed.
- All possible interventions performed by this app.

Dependency information involving a physiological parameter consists of two components: *effect* and *dependency type*. Effect of an intervention on a physiological parameter specifies the impact of the intervention on the parameter, which can be of three types as described below.

- *increase*: The parameter's value is increased by this intervention
- *decrease*: The parameter's value is decreased by this intervention
- *null*: The parameter's value is no affected by this intervention

Note that the effect can be both *increase* and *decrease*. The dependency type metadata can be either *primary* or *secondary*. We suggest the app developers to use the parameter names specified in the HumMod to describe the effects so that dependency metadata can be compared across apps. For example, if an app wants to perform an intervention that increases the heart rate, which causes a primary dependency, the app specifies the following dependency information in the app metadata.

```
<param> Heart-Rate.Rate </param>
<effect> increase </effect>
<dependency> primary </dependency>
```

To specify interventions, app developers are suggested to use the parameter names specified in HumMod. For example, an app that wants to administer 10 mg dosage of a drug called *Midodrine* specifies the intervention as shown below.

```
<intervention> MidodrineSingleDose.Dose </intervention>
<dosage> 10 </dosage>
```

Note that although HumMod is developed in C# and runs as an executable file (HumMod.exe), all the model parameters and their quantitative relationships are described in XML files. The executable file just parses the XML files and displays the results of the physiological simulations. Keeping the model parameters and their relationships in the XML files enables us to extend the model and enhance the model as medical field refines knowledge on human physiology.

6.7 EyePhy Dependency Checking

EyePhy considers all the specified parameters in the dependency analysis regardless of developer specified or caregiver specified. It also updates the list of unspecified parameters after performing the dependency analysis. EyePhy performs the following dependency checking at the installation time and at runtime.

6.7.1 Installation Time Checking

When a new app is installed, EyePhy performs dependency analysis across all other installed apps for *all the interventions* specified in the new app. The new app is considered conflicting if at least one of the following two conditions is satisfied when comparing one of its intervention with another intervention from an already installed app.

(a) There is an opposite effect on at least one parameter, or

(b) There is a same effect on at least one parameter, where the joint intervention can exceed the normal range of the parameter

Two interventions are considered to have an opposite effect on a parameter when one intervention increases it while another intervention decreases it and a same effect on a parameter when they both increase or decrease the parameter value. Condition (a) is checked by analyzing the effect metadata, i.e., keywords *increase*/*decrease* for the developer specified parameters, and by simulating the intervention using HumMod for the caregiver specified parameters (both high level and low level). To determine whether an intervention increases or decreases a parameter using HumMod, EyePhy monitors the parameter values for the *Window* duration with and without the intervention. If the parameter value with the intervention reaches two standard deviations higher (lower) than the corresponding value without the intervention for at least one observation, then EyePhy considers the effect is increase (decrease). The default value of the *Window* is 3 hours and can be changed if needed. Condition (b) is checked for all the high level parameters (regardless of specified or not) by simulating the joint intervention using HumMod and checking whether the parameter values stay within the acceptable range for the *Window* duration. However, EyePhy can't check for condition (b) for the

low level parameters as the normal ranges of these parameters are not known. If it finds that the new app can conflict with a previously installed app due to an intervention, it alerts the user with the detailed information and suggests using caution to install and use the most recent app. Otherwise, it allows the new app to get installed without such a notification.

6.7.2 Runtime Checking

At runtime, when an app requests to perform an actuation on the human body, that intervention request contains the suggested metadata related to the intervention. EyePhy checks whether the newly requested intervention can conflict with the previously granted interventions by checking for the conditions specified above in the installation time checking. However, there are two major differences from the installation time checking. First, EyePhy takes into account the time difference between the interventions at the runtime dependency analysis. To do that it uses the timestamps of interventions that are given input to EyePhy by the apps when they perform a direct actuation on the body or by the users when they listen to a suggestion, e.g., taking a drug or doing exercises. Second, at runtime, EyePhy maintains and updates the HumMod simulator based physiological model of the user by taking into account all the previously granted interventions. The apps that continuously monitor physiological parameters are allowed to provide input to EyePhy that is used to correct the physiological model. To determine whether the new intervention can conflict with the previously granted interventions, EyePhy checks whether the new intervention can cause any opposite effect with the most updated model (containing all the previous interventions) within the *Window* duration in at least one specified parameter using HumMod. It also checks whether the new intervention can cause any same effect with the most updated model and allowing the new intervention can exceed the normal range of at least one high level parameter, where the new intervention is started at the moment of request in the simulation.

An app's runtime intervention request contains a list of proposed interventions. EyePhy checks for conflicts for each intervention as mentioned above. It grants the first non-conflicting intervention in the list. If all the interventions in the list are conflicting, EyePhy doesn't grant the access of any intervention in the list. It notifies the user that the intervention request is denied and provides a brief explanation of why it is denied containing the parameter names where potential conflicts could occur. It also keeps a log of all the denied interventions to present to the caregivers and doctors. For each granted intervention, EyePhy updates a list of K unspecified parameters that are most significantly affected (highest % change from the without intervention value) by the intervention, which is later shown to the caregivers or doctors for additional consideration. Their chosen parameters are treated as caregiver specified parameters and used subsequent dependency analysis.

6.8 Evaluation

In this section, we demonstrate the conflict detection and resolution capability of EyePhy involving the human-in-the-loop apps. We describe a list of human-in-the-loop apps for the evaluation, low level parameters of interest, experimental setup, and installation time and runtime dependency detection capability of EyePhy.

6.8.1 App Selection

In order to evaluate EyePhy, we need a list of human-in-the-loop apps. Although there are many healthcare related apps in Apple App Store and Google Play, the apps usually provide health related tips and do not perform any direct actuation on the human body due to strict FDA regulation. Hence, we design the 20 human-in-the-loop apps for the evaluation as shown in Table 6.1. These apps are representative of the research work in the wireless sensor networks area and the available healthcare apps in smart phones. When designing these apps, we consider many types of human-in-the-loop apps including the ones that just monitor health conditions with no interventions, apps that perform direct actuations on the human body, apps that provide health related suggestions, and apps that perform actuation on the environment that affects the human body. Among thousands of medical conditions, we only choose a few of the medical conditions (between App #9 and App #14), because HumMod can directly simulate their effects on the human body. Drug descriptions and dosages are obtained from [121]. We only use the 12 apps (from App#9 to App#20) from Table 6.1 in the subsequent evaluation since the interventions performed by these apps can be simulated by HumMod. Other apps in Table 6.1 provide an overview about what else is proposed in the literature and available in the app markets. As HumMod supports more and more interventions, EyePhy will be able to perform dependency analysis across more apps.

6.8.2 Low Level Parameters of Interest

Based on the medical condition, specific low level physiological parameters need to be monitored for a user. In the evaluation, we consider three case studies focusing on three different parts of the human body.

1. 5 low level parameters related to the kidney: Kidney-ArcuateArtery.BloodFlow, Kidney-CO2.PCO2, Kidney-EfferentArtery.VasaRectaOutflow, Kidney-Lactate.Flow, and Kidney-Fuel.TotalGlucoseUsed(Cals/Min).

2. 5 low level parameters related to the heart: LeftHeart-BetaReceptors.Activity, LeftHeart-Flow.BloodFlow, LeftHeart-Flow.O2Use, LeftHeart-Flow.PO2Effect, and Heart-Ventricles.Rate.

3. 5 low level parameters related to the liver: Liver-O2.BloodFlow, Liver-Fuel.GlucoseDelivered(Cals/Min), ADH Clearance.Liver, GlycerolPool.LiverFARelease, and Liver-O2.InflowPO2.

ID#	App Name	Description
1	Lullaby	This app uses sensors to keep track of sleep quality of the user. It uses sound, light, temperature, and motion sensors to record the environmental condition during sleep. All these information is presented to the user in a tablet that helps him to identify the potential causes of sleep disruption.
2	Fall Detector	This app detects falls of the user using accelerometers and gyroscopes mounted on the body of the user. The app notifies the caregivers when a fall is detected.
3	Empath	This app monitors the activity levels, sleep quality, speech prosody, and weight of the user to detect depression. When a depression episode is detected, it turns on the lights and increases room temperature by 1 degree F of the occupied rooms.
4	Kintense	This app detects aggressive behavior of the user, e.g., hitting, kicking, pushing, and throwing using a Kinect sensor. When such a behavior is detected, it warns the medical stuff.
5	Musical Heart	This app monitors the heart rate and activity level of the user, and recommends music to help the user maintaining his target heart rate.
6	Food Nutrition	This app suggests eating nutritious foods. It also educates the user by providing nutrition facts of over hundreds of foods containing all the vitamins and minerals information.
7	Calorie Watcher	This app allows the user to set a daily calorie budget and suggests food recipe to maintain the budget. It also keeps a journal of all the food intakes.
8	Pollen Alert	This app suggests to stay home during the period when the pollen level is high outside.
9	Blood Pressure Control	This app treats high blood pressure and fluid retention caused by various conditions, including heart disease. It administers a drug called Chlorothiazide, which causes the kidneys to get rid of unneeded water and salt from the body into the urine. The app administers this drug once a day (1000 mg dosage) in the morning or two times a day (500 mg dosage) in the morning and in the late afternoon with meals.
10	Heart Rate Control	This app helps control the heart rate. It administers a drug called Digoxin, which is used to treat heart failure and abnormal heart rhythms. The app administers this drug once a day (0.5 mg dosage) in the morning or two times a day (0.25 mg dosage) in the morning and in the late afternoon.
11	Fluid Control	The app is used to reduce the swelling and fluid retention caused by various conditions, including heart or liver disease. It administers a drug called Furosemide, which causes the kidneys to get rid of unneeded water and salt from the body into the urine. The app administers this drug once a day (80 mg dosage) in the morning or two times a day (40 mg dosage) in the morning and in the afternoon.
12	Low Blood Pressure Control	This app treats low blood pressure condition. It administers a drug called Midodrine, which stimulates nerve endings in blood vessels, which tightens the blood vessels. As a result, blood pressure is increased. It is also used to treat dizziness that occurs upon sitting up or standing. The dosage is 10 mg, 3 times a day with at least 4 hours interval in between.
13	Glaucoma Control	This app treats glaucoma, a condition in which increased pressure in the eye can lead to gradual loss of vision. The app administers Acetazolamide, which decreases the pressure in the eye. Acetazolamide is also used to reduce the severity and duration of symptoms (upset stomach, headache, shortness of breath, dizziness, drowsiness, and fatigue) of altitude (mountain) sickness. Acetazolamide is used with other medicines to reduce edema (excess fluid retention) and to help control seizures in certain types of epilepsy. Dosage is 500 mg twice a day, one in the morning and one in the afternoon.
14	Blood Pressure Control II	This app treats patients with hyperaldosteronism (a condition when the body produces too much aldosterone, which is a naturally occurring hormone); low potassium levels; heart failure; and patients with edema (fluid retention) caused by various conditions, including liver, or kidney disease. The app administers Spironolactone, which is used alone or with other medications to treat high blood pressure. Spironolactone causes the kidneys to eliminate unneeded water and sodium from the body into the urine, but reduces the loss of potassium from the body. The app administers this drug once a day (200 mg dosage) in the morning or two times a day (100 mg dosage) in the morning and in the afternoon.
15	Insulin Injection	This app administers insulin based on the weight of the person and glucose in the blood level. Assuming the weight of the person is 171 lbs, it computes that the person needs 42 units of insulin everyday. It is delivered as a mixture of short acting and long acting insulins in 2 injections assuming the person has type-1 diabetes. Two thirds of the daily dosage is given before the breakfast and one third is given before the evening meal.
16	Activity Based HVAC Control	This app changes the room temperature based on the activities of daily living, e.g., while someone is eating, the app reduces room temperature by 1 degree F. Changing the ambient temperature may change the body temperature.
17	Humidity Control	This app turns on the humidifier to increase the humidity if the humidity goes below a threshold.
18	Physical Fitness	This app suggests to go out for exercise in the afternoon. The suggestion could be to do biking or to go to a gym to exercise on a treadmill.
19	Water Intake Monitor	This app monitors water intake in the body and suggests drinking more water if water intake is lower than a threshold.
20	Nutrition Intake Monitor	This app monitors nutrition intake in the body and suggests increasing nutrition intakes if needed.

Table 6.1: A List of Human-in-the-Loop Apps

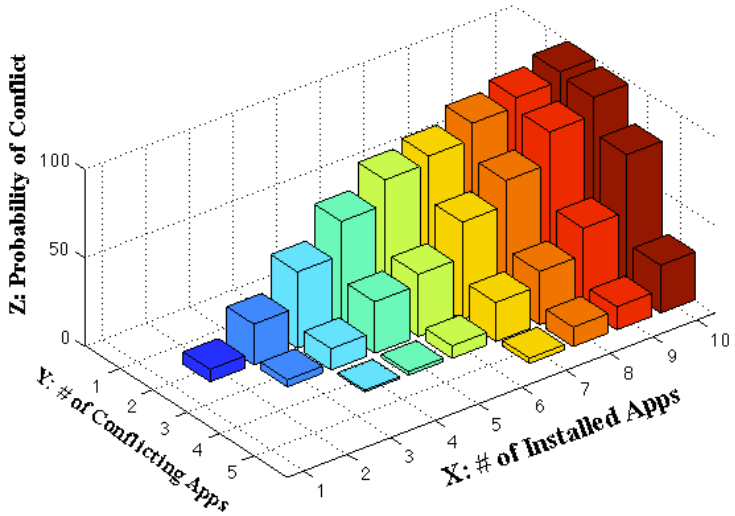
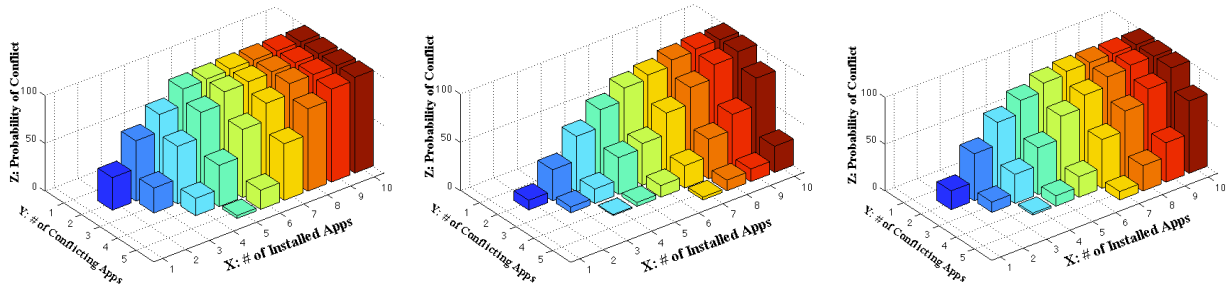


Figure 6.5: Installation time conflict detection on high level parameters



(a) Kidney (b) Heart (c) Liver

Figure 6.6: Installation time conflict detection on low level parameters related to the (a) Kidney, (b) Heart, and (c) Liver in addition to the high level parameters.

6.8.3 Experimental Setup

EyePhy offers personalized dependency analysis and hence it is expected that the user will provide some basic information including his gender, age, weight, and height to personalize the HumMod physiological simulator. For the evaluation, we assume that the user is a 37 years male having 171 pounds of weight and 178 cm height, which is the default configuration of HumMod. We determine the normal ranges of the high level parameters for an average healthy person of this age from [122]. For example, for this age, the normal blood pressure is between 90/60 mm/Hg and 120/80 mm/Hg, respiratory rate is between 12 and 18 breaths per minute, heart rate is between 60 and 100 beats per minute, and temperature is between 97.8 and 99.1 degrees Fahrenheit.

6.8.4 Static Analysis

At the time of app installation, EyePhy performs installation time check to detect potential conflicts with the previously installed apps by analyzing the dependency information specified in the app metadata. Figures 6.5 shows the probability of conflict between at least j apps when i apps are installed from the 12 apps (App #9 to App #20) in Table 6.1 using the

high level parameters only. Figure 6.6 shows the case when we consider the low level parameters in addition to the high level parameters. For each value x in the X axis, we randomly select x apps 100 times and compute the probability of conflict between at least y apps, where $1 \leq y \leq 5$ and $y \leq x$, and show the average results in these figures. Figure 6.5 shows that when someone installs 2 apps, there is just 7% probability of conflict between them if we just use high level parameters for conflict detection. However, Figure 6.6 shows that this probability becomes 34%, 10%, and 20% if we use low level parameters related to the kidney, heart, and liver, respectively in addition to the high level parameters. Hence, using low level parameters enables us to detect more potential conflicts in a personalized manner. We also see that installing more apps increases the probability of conflict. As we see in Figure 6.5, there is a 62% probability of conflict between at least 2 apps when someone installs 5 apps if we just use high level parameters. This probability becomes 98%, 75%, and 88% if we also use low level parameters related to the kidney, heart, and liver, respectively as shown in Figure 6.6. When someone installs 10 apps, the probability of conflict between at least two apps is 99% when we only consider the high level parameters. If we also consider any of the three low level parameters, the probability becomes 100% for each of the three cases. These results show the severity of conflicts involving high level and low level physiological parameters among the human-in-the-loop apps and demonstrate the need for a runtime system for detecting these conflicts.

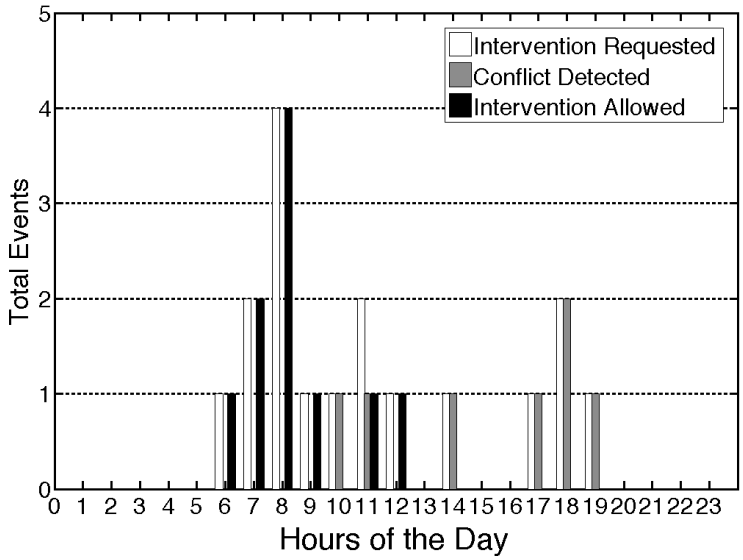
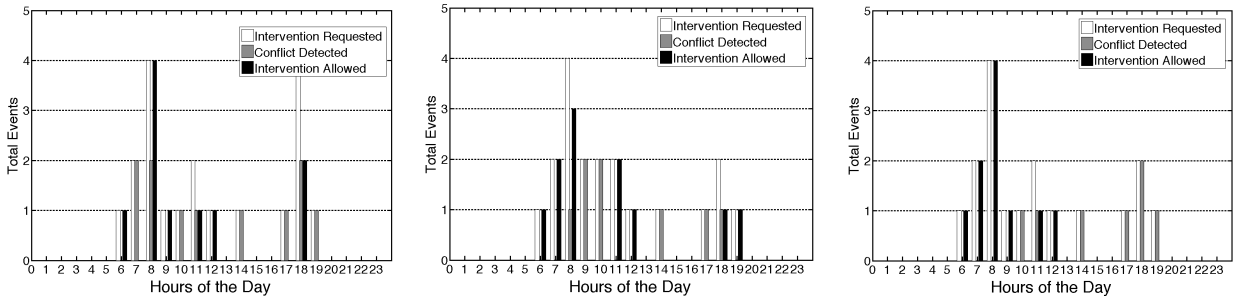


Figure 6.7: Runtime conflict detection on high level parameters

6.8.5 Runtime Analysis

In order to evaluate the runtime performance of EyePhy, we need to install some human-in-the-loop apps and let them perform interventions on one or more human subjects with and without EyePhy. However, doing simultaneous interventions from Table 6.1 on a human body may not be safe for the subject. Hence, instead of using real human subjects, we use HumMod to simulate the effect of interventions on the human body for the runtime analysis.



(a) Kidney (b) Heart (c) Liver

Figure 6.8: Runtime conflict detection on low level parameters related to the (a) Kidney, (b) Heart, and (c) Liver in addition to the high level parameters.

We assume that the user installs App #9 to App #20 in Table 6.1 in this evaluation. It may be questioned whether someone will install so many health related apps. We argue that more than 100,000 mobile medical apps are currently available [123], and in the Android app store itself, there are over 80 diabetes apps offering many functionalities including self monitoring of blood glucose recording, medication or insulin logs, and insulin dose calculator [124]. Although we do not have a statistic on the number of medical apps that people install in their smart phones, we see that average Android and iPhone users have 32 and 44 apps downloaded in their smart phones, respectively, based on a recent study [108]. Hence, it is not unreasonable to assume that some people will install a number of human-in-the-loop apps.

The way these apps (App #9 to App #20) perform interventions is described in Table 6.1. We run the experiment for a whole simulated day for an adult person using HumMod. The interventions depend on the lifestyle of the person. By default, in HumMod, the person sleeps from 10 PM to 6 AM and eats breakfast, lunch, and dinner between 7AM and 8 AM, 12 PM and 1 PM, and 6 PM and 7 PM, respectively. The apps that administer drugs take into account the meal times. App #16 (Activity based HVAC control) also takes into account the times of eating the meals. As we use this setting, the apps perform interventions based on the time of the day by considering the lifestyle of the person. We compute the number of intervention requests, the number of conflicts detected, and the number of allowed interventions by EyePhy in every hour during the day. We show the results in Figure 6.7 when considering the high level parameters and in Figure 6.8 when considering the low level parameters related to the kidney, heart, and liver in addition to the high level parameters. For brevity, we call these four cases as high-level, kidney, heart, and liver cases, respectively in the following description. The interventions take place in the following order in the four cases.

- At 6 AM, App #17 requests to perform an intervention to increase the room humidity, which is granted.
- At 7 AM, before the breakfast, App #15 requests to inject insulin. Also, at 7 AM App #16 requests to reduce room temperature by 1 degree F for the duration while the resident is having the breakfast. Both requests are granted in all the cases except the kidney case where we observe a conflict at the parameter Kidney-CO2.PCO2.

- At 8 AM, App #9, #10, #13, and #14 request to administer drugs. App #9, #10, and #14's intervention request contains a list of two dosages, one with a daily dosage and the other one with a half daily dosage. If the daily dosage amount is granted, the app does not perform any other intervention for the rest of the day. However, if the half dosage is granted, it performs another intervention in the afternoon (6 PM for App#9 and #10, and 7 PM for App #14) with the half daily dosage. If none of the dosages are granted at 8 AM, the apps retry the interventions at 9 AM, which may lead to administering a full daily dosage at 9 AM or two half daily dosages at 9 AM and 6PM/7PM. App #13 has just one dosage in the request that it tries to administer at 8 AM and at 7 PM. App #13 and #14's requests are granted in all the cases. App #9, #10's requests are granted in the high level case and the liver case. In case of the heart, App #9's request is granted, but App #10's request is denied because of a conflict at a heart related parameter LeftHeart-Flow.PO2Effect. In the case of kidney, App #9 and #10 both experience conflict for the daily dosage. However, both requests are resolved by using their half daily dosages that do not cause any conflict. The half daily dosage requests are granted.

- At 9 AM, App #11 requests for interventions, where the intervention list contains a daily dosage and a half daily dosage. If none of the dosages are allowed, App #11 retries at 10 AM. App #11's request is granted in all the cases except the case with the heart where we observe a conflict at the parameter LeftHeart-Flow.PO2Effect. In case of the heart, App #10 retries at 9 AM as its request at 8 AM is denied, which is denied again due to a conflict at a heart related parameter.

- At 10 AM, App #12 requests to perform an intervention, which is denied in all the cases due to a conflict with a high level parameter. In the case of the heart, App #11 retries at 10 AM, which is denied again due to conflict at a heart related parameter.

- At 11 AM, App #19 and #20 request to perform interventions. App #20's request is granted in all the cases. App #19's request is denied in all the cases except the case with the heart, where it is granted.

Based on the the way EyePhy is detecting and resolving the conflicts in all the four cases, it is evident that EyePhy's conflict detection and resolution can be personalized by focusing the analysis to specific low level parameters.

A summary of the total number of requested interventions, total number of conflicts detected, and total allowed interventions of the day are shown in Table 6.2. It shows that when considering the high level parameters, total 17 interventions are requested by the apps in a day, out of which 7 results in conflicts and the other 10 interventions are allowed. It also shows the results when specific low level parameters are considered in addition to the high level parameters. When low level parameters related to the kidney are monitored in addition to the high level parameters, 19 intervention requests are generated by the apps in a day. Among these requests, 11 requests are detected as conflicts and 2 of the 11 requests are resolved as described above. In addition to these 2 resolved conflicts, 8 other interventions are

Parameter Types	Total Requested IVs	Total Conflicts Detected	Total Allowed IVs
High Level Parameters	17	7	10
High level and low level parameters related to the kidney	19	11	10
High level and low level parameters related to the heart	19	8	11
High level and low level parameters related to the liver	17	7	10

Table 6.2: Summary of the interventions of 24 hours

allowed. It shows EyePhy's significant ability of detecting conflicts among interventions of apps at runtime as none of the conflicts could be detected without EyePhy. Note that detecting a single conflict involving human-in-the-loop can be crucial for making a life saving decision in some contexts.

6.9 Discussion

Although EyePhy takes into account a wide range of physiological parameters at the dependency analysis, the checking it performs with the parameters is relatively simple. The conflicting interventions it reports based on an *opposite effect* on a physiological parameter can be acceptable to some users due to their medical conditions. It does not check if the *same effect* causing interventions can exceed the acceptable range of any low level physiological parameter. In order to address these limitations, the physiological model needs to be more personalized, which requires advancement of medical research.

EyePhy uses HumMod to simulate the interventions. The simulation may incur non-negligible energy consumption if EyePhy is run on a smart phone. We develop and run EyePhy in a desktop computer as HumMod is not ported to any smartphone platform yet, where it takes about 2 seconds of real time to perform 10 minutes of simulation [73]. The runtime dependency analysis does not incur tremendous computational effort since we only need to do the simulation when an app requests for an intervention, which happens only about 20 times in a day in our evaluation. In the future, we will measure the energy consumption of EyePhy running in a smart phone. If the energy consumption makes it infeasible to use, we plan to move the HumMod simulation into the cloud.

6.10 Summary

Human-in-the-loop CPS apps pose new challenges for controlling human physiological parameters due to complex dependency issues of the apps' interventions. This chapter presents EyePhy, which is the first solution that directly addresses these issues. EyePhy runs as a service in the DepSys platform and uses a novel approach by employing a medically accepted physiological simulator that can model the complex interactions of the human physiology using over 7800 variables. It takes into account a wide range of physiological parameters at the time of dependency analysis as well as performs personalized dependency analysis. The end result permits a significant number of conflicts among the human-in-the-loop apps' interventions to be detected.

Chapter 7

Addressing Dependencies on Real-Time Constraints

The apps that DepSys runs at the top layer may have dependencies on real-time constraints. One crucial real-time constraint is to deliver packets reliably within latency bounds over wireless networks, e.g., an app may want to deliver control messages to turn on a breathing machine within a second. "Real-Time Delivery Scheduler" module in DepSys (shown in Figure 7.1) runs a service to address such dependencies on real-time constraints. In this chapter, we present a technique that DepSys uses for this purpose. This technique to deliver packets reliably and within latency bounds is applicable to real-time applications in other CPS areas, e.g., in large scale industrial plants, where sensors and actuators form a multihop wireless networks.

The rest of the chapter is organized as follows. Section 7.1 discusses the motivation of using the solution. Section 7.2 lists the technical contributions that this chapter makes. Section 7.3 formally defines the problem. Section 7.4 describes the model parameters, assumptions, and link classification based on an empirical study. Section 7.5 describes the scheduling algorithm that estimates latency bounds. Section 7.6 describes the experimental setup and the results of the experiments. Section 7.7 discusses some aspects of the solution and Section 7.8 describes a summary of the chapter.

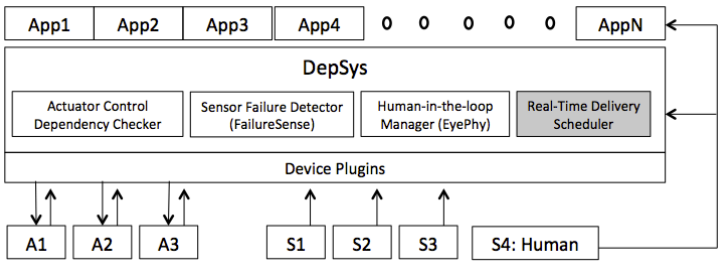


Figure 7.1: Real-Time Delivery Scheduler in DepSys

7.1 Motivation

The sensor and actuator networks in smart homes enable apps to perform fine grained sensing and to take sophisticated control actions. For the monitoring and control purposes, some real-time apps have dependencies on reliable and on time packet delivery. For example, a time-critical health-care app may require that the ECG data of a resident needs to be delivered to the platform where the app is running no later than 100 msec of the sensing time from the ECG sensors. A similar dependency is common in other CPS domains. For example, in chemical process control and industrial monitoring [125], distributed sensor networks are used to monitor and control the plant, where sensors and actuators form a multi-hop network. In such large scale settings, streams of data flow from one hop to another with a dependency on reliable and on time packet delivery. Traditionally, such applications use statically scheduled wired networks. However, the cost of wiring and re-wiring as the plant expands or shrinks is expensive. If a wireless sensor network can provide reliable real-time communication and on time delivery, the cost benefits and flexibility would be large.

However, it is very hard to provide reliable and on time packet delivery over wireless medium as wireless links are highly non-deterministic due to dependencies on two physical properties of the communication environment, link burstiness and interference. Link burstiness is a physical property which means that transmissions on a wireless link do not have independent probability of failure; instead they have periods of continuous message loss, i.e. they fail in a burst. Link interference is another physical property of the communication environment which causes packet transmission between different links to interfere with each other which results in packet loss. In order to provide reliability and latency bounds for packet delivery over wireless networks, we characterize these two physical properties and schedule packet transmission in a way that overcomes the difficulties offered by these properties of the communication environment. It is obvious that we cannot allocate only a single transmission time slot for a stream on each link, especially if we are dealing with bursty links. Because, if the transmission fails at that time slot due to a link burst, the node will need some additional time slots to transmit its packet. So, to provide the end-to-end latency bounds, we have to allocate more than one time slot per link for a stream. The number of time slots we need to allocate depends on the burstiness of the link. We don't want to allocate more time slots than we need, otherwise we will increase the latency bound. In addition, because of interference, other streams cannot be scheduled in nearby links during that entire multiple slot allocation time, which may increase the overall latency bound of all streams. So, achieving reliable communication and minimizing latency bound by scheduling is therefore a challenging goal.

The specific problem that we are addressing assumes that we are given a network topology and a set of periodic streams. For each stream, packets need to be delivered from the source node to its destination node within its period. "Real-Time Delivery Scheduler" module in DepSys (Figure 7.1) computes an upper bound of the latency for each of the

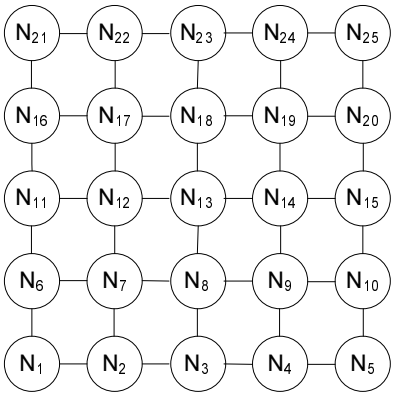


Figure 7.2: An Example Topology

streams by addressing the dependencies on link burstiness and interference. Current state of the art approaches can not bound latency, because most of the approaches use PRR based time slot allocation that fail to account for link bursts. Our 21 day long empirical study shows that over 23% links having PRR as high as 0.99 lose more than 50 packets in a row, some lose even over 1000 packets in a row. So, the PRR based approach can not produce a bound on latency. We need to carefully design some other metrics to characterize the burstiness of the links that will allow us to allocate sufficient number of time slots to produce a latency bound.

7.2 Contributions

This chapter makes three research contributions. First, to understand and characterize the dependency on link burstiness, we perform a 21 days of empirical study over a 802.15.4 network. Based on the findings, we define a new metric *max burst length (Bmax)* that allows us to classify links and allocate sufficient number of time slots to produce a latency bound of the streams. Empirical evidence shows that stationarity of link quality can be better characterized by $Bmax$ than PRR. Second, we design a static network-wide stream scheduling algorithm that uses a novel least-burst-route to produce latency bounds of the streams by addressing dependencies on link bursts and interference. Finally, by using testbed evaluation we show that we can actually bound the latency by achieving 100% packet delivery ratio within the derived latency bounds. We also investigate how the performance degrades when we do not have sufficient high quality links in the network.

An implication of these contributions is that if each of the streams has period greater than or equal to the latency bound that we provide, then our scheduling algorithm allows reliable communication subject to our burstiness and interference assumptions. If the burstiness characterization used for creating the schedules is violated during the course of execution, then a deadline might still be missed, but this is rare because the link characterization is performed

Stream	Period	ST	Source	Dest.	Route
S_1	20	1	N_1	N_4	N_1, N_2, N_3, N_4
S_2	20	1	N_2	N_5	N_2, N_3, N_4, N_5
S_3	20	1	N_7	N_9	N_7, N_8, N_9
S_4	10	1	N_{17}	N_{19}	N_{17}, N_{18},N_{19}

Table 7.1: An Example Set of 4 Streams

under realistic operating conditions and an adaptive solution will reduce such scenarios subsequently. Note that our approach does not minimize latency to maximize throughput. Instead, our average delivery latency is higher than most other techniques. However, we do offer a reliable communication and latency bound, which makes it easier to engineer predictable systems. We verify this claim by evaluating our approach on a 48-node wireless testbed with 10 simultaneous and periodic packet streams that shows our scheme has a 100% on-time delivery ratio.

7.3 Problem Definition

The problem that we are addressing is formally stated as follows: given a number of periodic streams and a network topology, calculate an upper bound on the latency of all the streams so that all the packets of all the streams reach their destinations within their respective latency bounds. In a home, the source nodes are usually the sensors and the destination nodes are usually the apps running on the DepSys platform. Also, there are cases where the source nodes are the apps running on the DepSys platform and the destination nodes are the actuators. Although the sensors and actuators in a home usually form a single hop network, our solution is generalized to support a multihop network, which is more common in large scale industrial plants. We assume stationary nodes and a fixed topology.

We define a Stream Set SS that contains n periodic streams, where a periodic stream S_i has a source node SRC_i, a destination node $DEST_i$, a route RT_i, a starting time ST_i and a period P_i, where i = 1, 2, 3, $\cdots$, n and a stream is represented as a 6-tuple (S_i,SRC_i, $DEST_i$, RT_i, ST_i, P_i). After the starting time is elapsed, at every period, the source node has a packet that has to be transmitted to the destination node by following that route.

Network topology is specified as a set of nodes N_1, N_2, N_3, $\cdots$, N_k along with their connectivity matrix and interference matrix. For each pair of connected nodes N_i and N_j, we denote the intermediate link as $L(i, j)$ that has burstiness parameters $Bmax$ and $B'min$ which are estimated based on empirical data. We will define these parameters formally in Section 7.4.

For example, consider the network topology shown in Figure 7.2. If two nodes are within their radio range, they are connected by an edge. Suppose that we are given 4 streams to calculate their latency bounds. The details of the streams are shown in Table 7.1. The route means stream S_1 is going from node N_1 to N_4 using route $N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_4$. So, we have SS = $\{(S_1, N_1, N_4, RT_1, 1, 20), (S_2, N_2, N_5, RT_2, 1, 20), (S_3, N_7$,$N_9$,

$RT_3, 1, 20), (S_4, N_{17}, N_{19}, RT_4, 1, 10)\}$ where RT_1 = $\{N_1, N_2, N_3, N_4\}$, RT_2 = $\{N_2, N_3, N_4, N_5\}$, RT_3 = $\{N_7, N_8, N_9\}$, RT_4 = $\{N_{17}, N_{18}, N_{19}\}$.

The apps that have dependencies on real-time constraints provide the relevant streams' information to the "Real-Time Delivery Scheduler" module is DepSys (Figure 7.1) before starting the flow of the streams. "Real-Time Delivery Scheduler" module uses this information along with the topology information to schedule all the streams and to output latency bound LB_i for each stream S_i. We claim that every packet of stream S_i will reach its destination no later than $ST_i + LB_i$ if packet transmission is performed according to our schedule.

7.4 Empirical Study

To see how packet transmission is dependent on link burstiness, we run a 21 days long experiment on a 48 node 802.15.4 network and transmitted 3,600,000 packets over every link. From these experiments we confirm that burstiness in wireless packet transmission is a ubiquitous phenomenon. We also observe that for links with similar link quality, their burstiness behavior can be different. As a result, the transmission latency of data streams depends on burstiness of links that the streams go through.

7.4.1 Model Parameters and Assumptions

To characterize link bursts we define five parameters: B, $Bmax$, B', $B'min$, and W for every link. The way these parameters are computed and defined is as follows: after transmitting 3,600,000 packets over every link, we have a long sequences of data trace of 0s and 1s per link where 1 at ith index of the sequence means that packet with ith sequence number was successfully transmitted and 0 at that place means it failed. As an example, consider a sample data trace 0110010011 of length 10. Now, we define W as a window for packet transmission having length $|W|$. For this particular example data trace, assume that we have $|W|$ = 3. So, we have 8 different windows to consider, each of length 3, where the first window spans from index 1 to 3 having values 011, the second window spans from index 2 to 4 having values 110, the third window spans from index 3 to 5 having values 100 and so on. For a particular window, we define B as the number of time slots where packet transmission failed due to link bursts and B' as the number of time slots where packet transmission was successful. So, for this particular example, we have B = 1, B' = 2 for the first window, B = 1, B' = 2 for the second window, B = 2, B' = 1 for the third window and so on. Now, for a particular window of size $|W|$, we define $Bmax$ as the maximum value of B for all possible windows of size $|W|$ and $B'min$ as the minimum value of B' for all possible windows of size $|W|$. So, for this particular example, $Bmax$ = 2 and $B'min$ = 1 where $|W|$ = 3.

So, the key idea is, we define $Bmax$ as the maximum number of time slots where packet transmission can fail due to a burst and $B'min$ as the minimum number of time slots that are available for packet transmission between

two consecutive bursts. So in a window size of $Bmax + B'min$, we have at least $B'min$ time slots for successful packet transmission. Note that different links have different burst characteristics and to obtain a particular $B'min$ we need to consider different sized window for different links. *So, we are not defining a window for all the links, instead we are calculating the window size of a link for a particular $B'min$ by calculating first $Bmax$ and then computing $|W| = Bmax + B'min$.*

Algorithm 2 : ComputeBmax(D, $B'min$)

```
1:  for i ← B'min + 1 to |D| do
2:    isSatisfied ← TRUE
3:    for j ← 1 to |D| − i do
4:      B' ← Σ_{k=j}^{j+i−1} D[k]
5:      if B' < B'min then
6:        isSatisfied ← FALSE
7:        break
8:      end if
9:    end for
10:   if isSatisfied = TRUE then
11:     Bmax ← i − B'min
12:     return Bmax
13:   end if
14: end for
15: return −1
```

Algorithm $ComputeBmax()$ computes $Bmax$ given a data trace D of 0s, 1s and $B'min$. It returns -1 if the data trace doesn't have a $Bmax$ that satisfies the condition specified by the definition of $Bmax$ amd $B'min$. The running time of the algorithm is $O(|D|^2)$ which is large for long data trace, although B' at line 4 can be computed in $O(1)$ time by using a dynamic programming based memoization approach. But if a link has a very high $Bmax$, it indicates that the link is prone to heavy bursts, and we avoid this link for real time applications. So, for practical purposes, we limit the maximum $Bmax$ to be $C = 1200$, and constrain the loop at line 1 to run from 1 to C. Then, the running time of the algorithm becomes $O(C|D|)$ which is linear with respect to the size of the data trace.

Now we describe the necessity and importance of using $Bmax$ for calculating the latency bound. Some links do have arbitrarily large $Bmax$ and such links are avoided in choosing routes in our methodology. So, we are only considering links having $Bmax \leq 1200$. Figure 7.3 demonstrates the $Bmax$ of the links (when $B'min = 1$) for different values of PRR. The figure shows that some links have large $Bmax$ although the corresponding PRR is as high as 0.99. Our experimental result shows that we have 32.72%, and 12.32% links having PRR as high as 0.99 and 0.999 suffer from a burst of having length ≥ 5. So, if we allocate time slots based on PRR, some packets may not get through due to a burst of "unexpected" length and the packet will miss its deadline. So, $Bmax$ is a better metric for allocating time slots to offer a latency bound compared to PRR as it is considering link bursts.

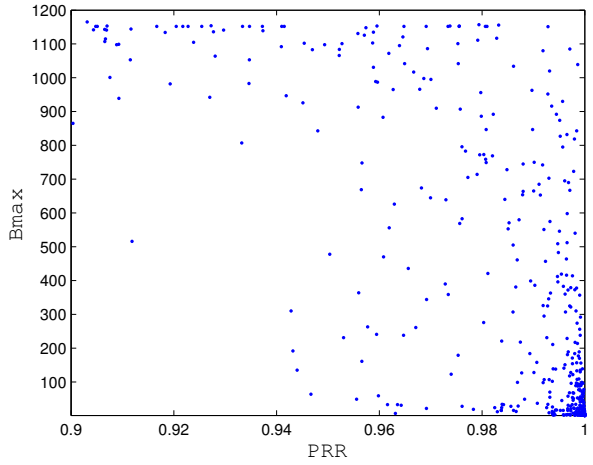


Figure 7.3: $Bmax$ vs. PRR

To demonstrate that links having low $Bmax$ tend to be more stationary and $Bmax$ can characterize non-stationary links better than PRR we present Figure 7.4 and Figure 7.5. We calculate $Bmax$ of the links after 14 days and 21 days and plot the percentage of the links having stationary $Bmax$ within this interval. Figure 7.4 shows that low $Bmax$ links tend to be more stationary. In contrast, we compute PRR of the links after 14 days and 21 days and plot the percentage of links having stationary PRR within this interval. We consider from links having PRR 0.98 and plot with an increment of 0.0005 in Figure 7.5 that shows that links having PRR as high as 0.99 don't preserve stationarity even over seven days.

Our model assumes that after network characterization, i.e., after we compute $Bmax$ and $B'min$ of every link, if we consider $Bmax + B'min$ time slots for packet transmission, we have at least $B'min$ time slots to transmit packet successfully. Although the assumption looks questionable, we have some strong arguments in favor of it. The assumption may not hold in a battlefield where link behavior may change drastically, but it seems to hold in a regular working environment like offices, universities, and industrial plants if we can characterize the links for a long period of time under all possible working environments. Obviously, this assumption will not hold for all the wireless links. We classifiy the links (in the next section) for which the assumption seems to be true. If the wireless links are not good enough to meet the assumption, we can move some nodes or add additional nodes in the network to create the "right" topology having good burst properties that satisfies the model assumption. Also, recent work [126] has explored that bursts in wireless link have scaling properties meaning that the bursts shows self-similarity or other coherent structure over many time scales without having long range dependence. The paper specifies an onset point of 640 ms where random variations stop affecting the wireless link and self-similarity starts to dominate. It clearly indicates the possibility of capturing the burstiness characteristics of the wireless links if we investigate the burstiness behavior of the wireless links over a long period of time.

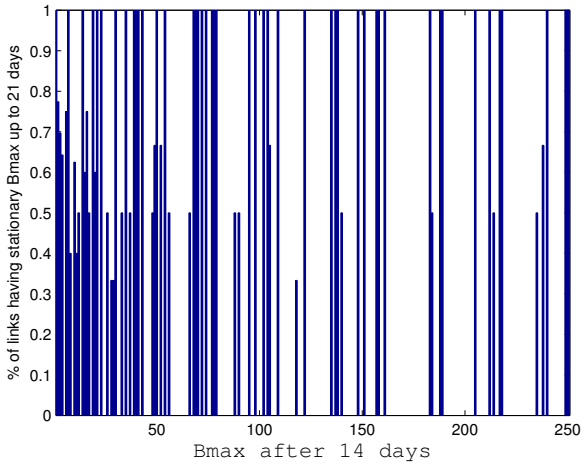


Figure 7.4: $Bmax$ as a Stationarity Metric

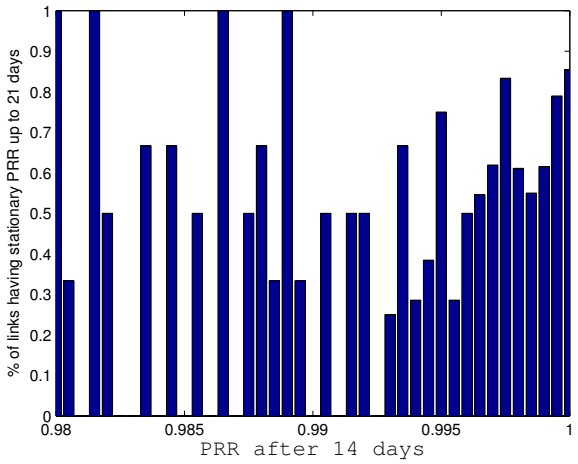


Figure 7.5: PRR as a Stationarity Metric

To characterize link interference, we define an interference matrix, IM that represents the interference pattern for the network by specifying which links potentially interfere with others.

7.4.2 Link Classification

Based on the values of $Bmax$, we can classifiy the wireless links into 4 categories:

1. Stationary Links
2. Asymptote-Stationary Links
3. Epsilon-Stationary Links
4. Non-Stationary Links

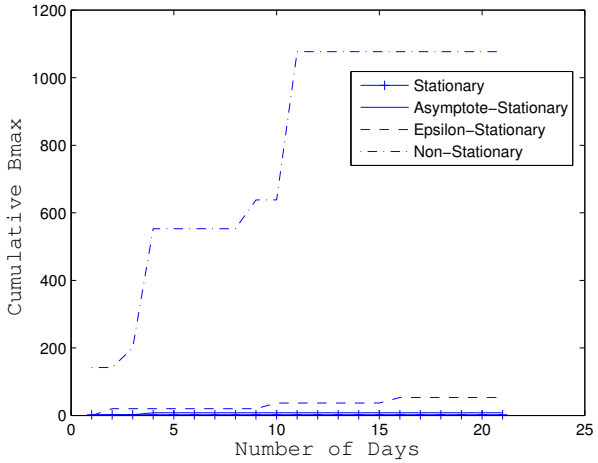


Figure 7.6: $Bmax$ vs. Days

$Bmax$	Links	Application
Small	Stationary, Asymptote-Stationary	High Priority Hard Real Time Applications
Large	Stationary, Asymptote-Stationary	Low Priority Hard Real Time Applications
Small	Epsilon-Stationary	High Priority Soft Real Time Applications
Large	Epsilon-Stationary	Low Priority Soft Real Time Applications
Small/ Large	Non-Stationary	Other Applications

Table 7.2: Classification of Links for Different Applications

We calculate $Bmax$ of the links for every day, and calculate the $cumulativeBmax$ which is the maximum $Bmax$ from the first day up to that day. Figure 7.6 shows how the $cumulativeBmax$ varies as the days proceed and allows us to distinguish between Stationary links and Asympote-Stationary links. We call the links having constant $cumulativeBmax$ from the first day as Stationary links. We call the links Asymptote-Stationary for which the $cumulativeBmax$ becomes constant after a few days (we set this threshold as 14 days). Epsilon Stationary and Non-Stationary links do not show stable $cumulativeBmax$ even after 14 days and they are largely affected by the physical environment. The probability of observing large bursts is very small for both cases, but in case of Epsilon Stationary links we have not observed any burst of having length > 1000 while in case of Non-Stationary links we have observed that within 21 days.

Figures 7.6 is not only important for link classification, it is also essential for calculating $Bmax$ that the scheduler needs to know to allocate time slots. We can only use Stationary and Asymptote-Stationary links for hard real time applications where the corresponding $Bmax$ will be computed from the asympotes of the curves from Figure 7.6. Epsilon-Stationary links can be used for soft real time applications with some probability of missing end-to-end deadline. For example, assume that an application requires 99% end-to-end packet delivery on time. The probability calculation may proceed from the distribution of B as follows: all we have to do is to find a minimum $Bmax$ that will allow us not to miss end-to-end deadline more than 1% of the time. If the route is n hop, then for every link L_i that the route goes

through can have a $Bmax = b_i$ where the probability of observing a burst having length greater than b_i is less than or equal to $\sqrt[n]{0.01}$ i.e. $Prob[Bmax > b_i] \leq \sqrt[n]{0.01}$. Note that this approach allows us to schedule packet transmission based on QoS where an application can specify its requirement by quantifying the end-to-end deadline miss raio it can afford. But this approach assumes that bursts between links are independent which is not yet validated. Hence, we will consider the use of Epsilon-Stationary links in our future work. Table 7.2 shows how the classification of the links can be useful to select links for different application purposes.

7.5 Solution

To explain our solution we first present some preliminary discussion. In Section 7.5.1, using a simple example that assumes a single stream we show how to deal with end-to-end latency in the presence of burstiness. In Section 7.5.2, we expand the example to multiple streams where both burstiness and interference must be handled. After these preliminary discussions we present the complete algorithm for latency bound generation.

7.5.1 Dealing with a Single Stream

If there is only one stream in the network and there is no packet loss, then the end-to-end latency bound is the sum of per link latencies in all the intermediate links. This is the theoretical lower bound of end-to-end latency for a single stream. But in reality, links are not ideal and packet losses occur in a burst. If a stream has n intermediate links from source to destination and ith intermediate link has burstiness parameters $Bmax = b_i$ and $B'min = b'_i$ ($i = 1, 2, 3, \cdots, n$), then on ith link we have to allocate $b_i + 1$ time slots for the stream. So the end-to-end latency bound LB is given by Equation (7.1).

$$LB = \sum_{i=1}^{n} (b_i + 1) \tag{7.1}$$

Consider the topology in Figure 7.2. Assume that the links have $Bmax$ and $B'min$ as shown in Table 7.3.

Link	$Bmax$	$B'min$
$L(1,2)$	2	2
$L(2,3)$	3	2
$L(3,4)$	3	3
$L(4,5)$	3	2
$L(7,8)$	2	2
$L(17,18)$	2	3
$L(18,19)$	1	4

Table 7.3: $Bmax$ and $B'min$ of Links

Now, assume that our SS consists of a single stream S_1 from Table 7.1 having period 20, starting time slot 1 and it goes from N_1 to N_4 using route $N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_4$. So, $SS = \{(S_1, N_1, N_4, RT_1, 1, 20)\}$ where $RT_1 = \{N_1, N_2, N_3, N_4\}$.

Link/TimeSlot	1	2	3	4	5	6	7	8	9	10	11
$L(1,2)$	S_1	S_1	S_1								
$L(2,3)$				S_1	S_1	S_1	S_1				
$L(3,4)$								S_1	S_1	S_1	S_1

Table 7.4: Scheduling Table with a Single Stream

Since, $Bmax$ is 2 for $L(1,2)$, if we allocate $(Bmax + 1)$, i.e., 3 time slots for S_1 at node N_1, then the packet will reach to N_2 even if there is a burst. Similarly, if we allocate 4 time slots at N_2 to transmit over $L(2,3)$ and 4 time slots at N_3 to transmit over $L(3,4)$ then it takes only $3 + 4 + 4 = 11$ time slots to ensure that every packet of S_1 will be delivered to its destination within that time even if there is a burst in a number of links. Hence, $LB_1 = 11$. The corresponding scheduling table is shown in Table 7.4 below where the column represents time slots and the row represents links. It shows which stream will be transmitted at which time slot using which link. For example, S_1 will be transmitted at time slots 1, 2 and 3 using link $L(1,2)$.

Note that multiple time slots are reserved to ensure reliability. It does not mean that N_1 is transmitting three packets of S_1 in time slots 1, 2 and 3. Rather it means that N_1 will try in these time slots to transmit a packet of S_1 and it will stop as soon as the packet gets in to the next hop node, i.e., N_2. We assume that the radio transceiver supports hardware/software acknowledgements so that the receiver can acknowledge its packet reception immediately to the sender. This is a reasonable assumption based on the radio transceivers available in the market.

7.5.2 Dealing with Multiple Streams

When we have only one stream in the network, even in the presence of burstiness, the scheduling is not complex since the interference is not an issue. No two links can interfere with the transmission of each other in this scenario. But when multiple streams co-exist in the network, we have to take interference into account and ensure that no two interfering links transmit packets at the same time slot causing a pacekt loss. Note that finding the minimum average latency bound of all the streams by scheduling packet transmission considering all possible routes for every streams subject to link interference and link burstiness is a NP-Hard problem since it is analogous to the bin packing problem [127]. Hence we use a greedy solution based on the following principles:

1. Schedule packet transmission up to the LCM of the periods of all the streams. If all the streams are schedulable within the LCM of their periods, we conclude that the stream set is schedulable and we can offer a latency bound for every stream.

2. Address packet loss due to link interference as follows: First, figure out the interference pattern of the network, represented by IM empirically. Then, schedule packet transmission in a way to make sure that no two interfering links 'transmit' packets at the same time slot.

3. Address packet loss due to link burst as follows:

   (a) Allocate $Bmax + 1$ contiguous time slots for packet transmission over a link. Note that different links have different $Bmax$. We are assuming that the route is *least-burst-route* as the route is the shortest path having the minimum sum of $Bmax$ from the source node to the destination node.

   (b) While allocating $Bmax + 1$ time slots, overlap at most $B'min$ streams' time slot allocation. The reason is, within a window of $Bmax$ + $B'min$, there are at least $B'min$ good slots for packet transmission, and so, we should be able to transmit at least $B'min$ number of streams. Hence, we allow slots of at most $B'min$ streams to overlap. There are two conditions for it: (i). This overlapping is allowed only when packets are being transmitted over the same link. Note that we can not overlap time slots for neighboring nodes due to link interference. (ii). While overlapping time slots of multiple streams, no two streams are allowed to be allocated the same $Bmax + 1$ time slots, i.e., complete overlapping is not allowed. The reason is, if two streams S_1,S_2 are allocated the same $Bmax + 1$ time slots, and if we lose $Bmax$ time slots due to a burst, then we can not transmit packets of both S_1 and S_2 in the remaining time slot.

Link/TimeSlot	1	2	3	4	5	6	7	8
$L(1,2)$	S_1	S_1	S_1	S_1				
$L(1,2)$					S_2	S_2	S_2	S_2

Table 7.5: Schedule without Overlapping

Link/TimeSlot	1	2	3	4	5
$L(1,2)$	S_1	S_1, S_2	S_1, S_2	S_1, S_2	S_2

Table 7.6: Schedule with Overlapping

To explain why overlapping is important and how it is done, consider the following example. Assume that Stream Set SS consists of two streams S_1, and S_2, both going from node N_1 to N_2 using route $N_1 \rightarrow N_2$ through the link L(1,2). Assume that both streams have period 20 and starting time slot 1. Also assume that $Bmax = 3$ and $B'min = 2$ for the link L(1,2). A schedule without overlap is shown at Table 7.5 where average Latecy Bound per stream is $(4 + 8)/2 = 6$ time slots. Compare it with a schedule with overlapping in Table 7.6 where average Latency Bound per stream is $(4 + 5)/2 = 4.5$ time slots. Note that we could not do complete overlapping as in Table 7.7. Because, as it is mentioned earlier, if we lose $Bmax = 3$ time slots due to a burst, then we can not transmit packets of both S_1 and S_2 in the remaining time slot.

Link/TimeSlot	1	2	3	4
$L(1,2)$	S_1, S_2	S_1, S_2	S_1, S_2	S_1, S_2

Table 7.7: Schedule with Complete Overlapping

Link/TimeSlot	1	2	3	4	5	6
$L(1,2)$	S_1	S_1, S_2	S_1, S_2, S_3	S_2, S_3, S_4	S_3, S_4	S_4

Table 7.8: Schedule with Maximum Overlapping

Overlapping time slots raises another issue. Since we can transmit only one packet at a time and the packet transmission process is completely deterministic, we have to prioritize the streams to be transmitted in the overlapped time slots. Our *prioritizing rule* works as follows: if multiple streams are scheduled to be transmitted at the same time slot, transmit the not-yet-transmitted stream that has the closest ending time slot. We will see its use in the next example.

To illustrate how many streams can be overlapped consider the following example. Assume that Stream Set SS consists of four streams S_1,S_2,S_3, and S_4 all going from node N_1 to N_2 using route $N_1 \rightarrow N_2$ through the link L(1,2). Assume that all the streams have period 20 and starting time slot 1. Also assume that $Bmax = 2$ and $B'min = 4$ for the link L(1,2). Table 7.8 demonstrates a schedule that shows within a window of size $Bmax$ + $B'min$ = $2 + 4$ = 6, we can overlap at most $B'min$ = 4 streams. This schedule will always work if nodes transmit packets according to the prioritizing rule. For example, if packet transmission fails during time slots 1, and 2, packets of streams S_1,S_2,S_3, and S_4 will be transmitted at time slots 3,4,5 and 6, respectively. If packet transmission fails at time slots 2, and 4 then packets of streams S_1,S_2,S_3, and S_4 will be transmitted at time slots 1,3,5 and 6. So, even if packet transmission of any combination of size $Bmax$ fails within a window of $Bmax$ + $B'min$, all the packets of all the streams will get through to the next node if we allocate time slots according to the principles mentioned earlier and nodes transmit packets according to the prioritizing rule as it is proved in the next subsection.

7.5.3 Correctness Proof

The correctness of our algorithm depends on the following theorem:

Theorem: *If we overlap packet transmissions of at most b' streams in $(b + b')$ time slots, all going through the same link having $Bmax = b$, $B'min = b'$, each stream having $(b + 1)$ contiguous time slots allocated without complete overlapping with any other stream, then all of the streams will get through even if there is a burst of at most b time slots (not necessarily contiguous), if we transmit packets according to our prioritizing scheme.*

Proof: The proof is by contradiction. For a contradiction assume that stream S_i could not be transmitted due to a burst. Since we can lose at most b time slots due to a burst, there has to be a time slot, we call "good slot", when there was no burst, but still S_i was not transmitted. The reason for that can only be some other stream S_j was transmitted at

that time slot. Note that the good slot can not be the last sending time slot of S_i. Because, at that time slot, steam S_i has highest priority for transmitting packet. So, no other stream S_j can be transmitted at that time slot. So, now we have to consider one remaining possibility. There was a burst at the last sending time slot of S_i and among the previous b slots of S_i, $(b-1)$ slots are gone due to a burst leaving one good slot and at that time slot some other stream S_j was transmitted for the priority scheme. Note that this can not happen either, because as S_j has higher priority than S_i, S_j has started time slots before S_i did, and hence it should be transmitted even before the burst happens. Because, we are overlapping at most b' streams in any time window of $(b+b')$ slots and there has to be at most b' good slots between any consecutive bursts. So, S_j will be transmitted in the good time slots of the previous window and S_i will be transmitted within this time window. So, there is no way S_i will not be transmitted due to a burst.

7.5.4 Algorithm Details

Algorithm 3 schedules packet tranmission up to the LCM of the periods of the streams based on the principles and conditions stated in Section 7.5.2. It takes the Stream Set SS of n streams (as defined in Section 7.3), topology, Interference Matrix IM, burstiness parameters $Bmax$, $B'min$ of every link as inputs and returns either $true$ if all the streams are schedulable or $false$ if they are not schedulable. If all the streams are schedulable, then LB_i holds the latency bound of stream S_i.

Let L_i be the last allocated time slot for S_i in the scheduling algorithm. So, initially we have L_i = (ST_i - 1) (at line 4) for every stream S_i. We need to adjust the starting time ST_i of stream S_i when its period P_i is over (at line 40) to correcly compute the latency bound LB_i (at line 29). Let, N_i be the node that has received the last transmitted packet of S_i. So, initially $N_i = SRC_i$ (at line 5) which is the source node of stream S_i. Assume that N_{i+1} is the next node to which N_i has to transmit a packet of S_i. Note that N_{i+1} can be easily determined from the route of S_i.

We put $DeferThreshold = 2$ (used at line 20). If you use a higher value for it, the scheduler runs faster, but the generated latency bound may also rise. We are deferring time slot allocation to only those streams that can not take advantage of overlapping time slots with other streams. The reason for deferring time slot allocation is to increase parallelism over the link from N_i to N_{i+1}. This is a kind of lazy approach for allocating, because we are hoping that it is possible that some other streams may show up and can be allocated in these time slots that can exploit parallelism.

Table 7.9 and Table 7.10 show the whole scheduling for the example problem we are working with from Section 7.3 and B, B' of Table 7.3. Here S_1 and S_2 share time slots at links $L(2,3)$ and $L(3,4)$. The latency bound of the streams is shown in Table 7.11.

Our algorithm exploits parallelism in two ways. The first one is, multiple streams can be transmitted at the same time if there is no interference in their transmission as it is seen in S_1 and S_4 at time slots 1, 2 and 3 in Table 7.9. The

Algorithm 3 : Scheduler(SS, topology, IM, $Bmax$, $B'min$ of every link)

```
1: n ← |SS|
2: lcm ← LCM(P_1, P_2, P_3, ... P_n)
3: for i ← 1 to n do
4:    L_i ← ST_i − 1,
5:    N_i ← SRC_i,
6:    LB_i ← 0
7: end for
8: for t_1 ← 1 to lcm do
9:    if All the streams are scheduled then
10:      break
11:   end if
12:   for i ← 1 to n do
13:      if S_i is already scheduled then
14:         continue
15:      end if
16:      if t_1 = L_i then
17:         for t_2 ← t_1 + 1 to ST_i + P_i do
18:            (b, b') ← (Bmax, B'min) of link L(N_i, N_{i+1})
19:            if it is possible to allocate (b + 1) contiguous time slots starting from t_2 by following the principles and
               conditions then
20:               if S_i is not making any overlap with any streams in these time slots AND (t_2 − t_1) > DeferThreshold
                  then
21:                  L_i ← t_2 − 1
22:                  break
23:               else
24:                  Allocate these (b+1) time slots for S_i in N_i
25:                  L_i ← t_2,
26:                  N_i ← N_{i+1}
27:                  if N_{i+1} is the destination node for S_i then
28:                     Consider that S_i is scheduled
29:                     LB_i ← max(LB_i, t_2 + b + 1 − ST_i)
30:                  end if
31:                  break
32:               end if
33:            end if
34:         end for
35:      end if
36:      if mod(t_1, P_i) = 0 then
37:         if S_i is not scheduled yet then
38:            return false
39:         else
40:            ST_i ← ST_i + P_i,
41:            N_i ← SRC_i
42:         end if
43:      end if
44:   end for
45: end for
46: if all the streams are not scheduled within the lcm then
47:   return false
48: end if
49: return true
```

Link/TimeSlot	1	2	3	4	5	6	7	8	9	10
$L(1,2)$	S_1	S_1	S_1							
$L(2,3)$				S_1	S_1,S_2	S_1,S_2	S_1,S_2	S_2		
$L(3,4)$									S_1	S_1,S_2
$L(4,5)$										
$L(7,8)$										
$L(8,9)$										
$L(17,18)$	S_4	S_4	S_4							
$L(18,19)$				S_4	S_4					

Table 7.9: Scheduling Table with Multiple Streams: Part 1

Link/TimeSlot	11	12	13	14	15	16	17	18	19	20
$L(1,2)$										
$L(2,3)$										
$L(3,4)$	S_1,S_2	S_1,S_2	S_2							
$L(4,5)$				S_2	S_2	S_2	S_2			
$L(7,8)$				S_3	S_3	S_3				
$L(8,9)$								S_3	S_3	S_3
$L(17,18)$	S_4	S_4	S_4							
$L(18,19)$				S_4	S_4					

Table 7.10: Scheduling Table with Multiple Streams: Part 2

Stream	Latency Bound
S_1	12 i.e. max$(12-1+1,0)$
S_2	17 i.e. max$(17-1+1,0)$
S_3	20 i.e. max$(20-1+1,0)$
S_4	5 i.e. max$((5-1+1),(15-11+1),0)$

Table 7.11: Latency Bound for Each Stream

second way is, when two streams are going over the same link, they can share at most $Bmax$ time slots as it is observed between S_1 and S_2 in time slots 5, 6 and 7 in Table 7.9.

The running time of the algorithm is $O(LP)$ where L is the LCM of the periods of the streams and P is the sum of the periods of the streams. The reason is, the for loop at line 8 takes $O(L)$ time and the for loops at lines 12 and 17 together takes $O(P)$ time. Note that $O(LP)$ depends mainly at the periods of the streams, and it can be exponential if the streams have periods that are prime numbers.

7.6 Evaluation

In this section, we evaluate our algorithm in terms of end-to-end deadline miss ratio and latency bound. At first we describe the experimental setup. Then we describe how we measure burstiness parameters $Bmax$ and $B'min$ of every link and the interference matrix, IM. Then the effect of $Bmax$ and $B'min$ to end-to-end deadline miss ratio and latency bound is evaluated.

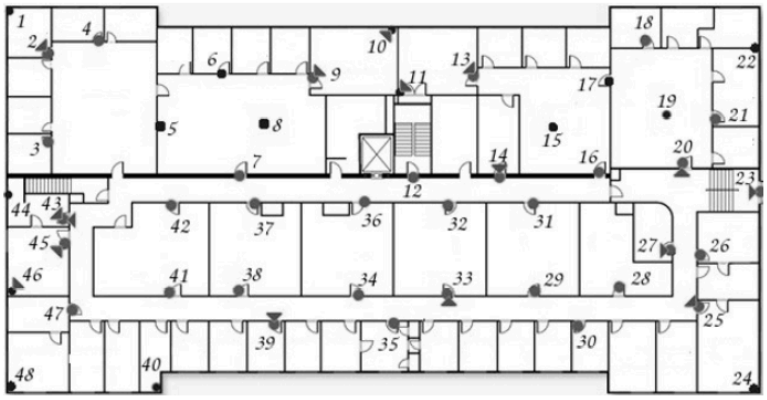


Figure 7.7: Testbed Layout

7.6.1 Experimental Setup

As mentioned in Section 7.4, we run a 21 day long experiment to understand how burstiness affects packet transmission in an indoor testbed. This testbed consists of 48 TMotes that use the ChipCon CC2420 radio. These nodes are deployed on walls and ceilings of a building, as shown in Figure 7.7.

7.6.2 Measuring Burstiness

To understand the effect of link burstiness on packet transmission and to calculate the model parameters $Bmax$ and $B'min$ of every link, we transmit 3,600,000 packets per link. The packet transmission rate is around 200 packets per second where every node tries to transmit the next packet as soon as possible with a zero inter packet interval. To avoid possible collisions and interference, we allow only one node to transmit packet at a time. There are 48 nodes, each node takes turn for packet transmissoin and at each turn a node transmits around 1200 packets continuously. We are supposed to transmit $21 \times 24 \times 60 \times 60 \times 200/48 = 7560000$ packets per link in 21 days with the specified packet transmission rate. But we could only transmit 3,600,000 packet per link, because it takes time to fetch the sequence number of the received packets from all other nodes when a node finishes its turn of packet transmission. After collecting sequence numbers of received packets on every link we have a long data trace that is used to calculate model parameters $Bmax$ and $B'min$ of every link using algorithm 2. The spatial distribution of $Bmax$ of the links at the testbed is shown in Figure 7.8, where green zones represent links having low $Bmax$, and red zones represent links having high $Bmax$. We find that most of the red zones fall within places where peoples' movement varies a lot, e.g., stairs, restrooms, copy-room, conference rooms, and kitchen.

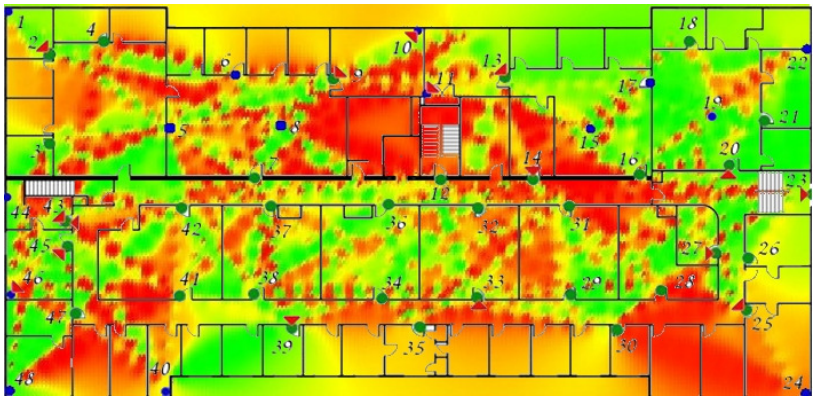


Figure 7.8: Spatial Distribution of $Bmax$

7.6.3 Measuring Interference

The external inteference, e.g. interference caused by WiFi networks, is captured through the $Bmax$ and $B'min$ parameters. To address internal interference, i.e. interference caused by packet transmission through neighboring links at the same time, we define a $k \times k$ interference matrix, IM for a network of k links that specifies which links potentially interfere with others:

$$IM(i,j) = \begin{cases} 1 & \text{if link } L_i \text{ and link } L_j \text{ are in interference range,} \\ 0 & \text{otherwise.} \end{cases} \tag{7.2}$$

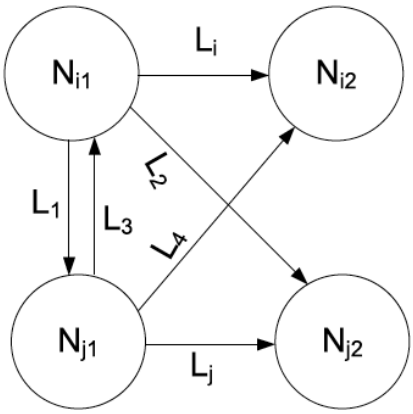


Figure 7.9: Measuring IM

If $IM(i,j) = 1$, then scheduler will make sure that two packets are not scheduled for transmission over links L_i and L_j at the same time. The computation of IM is based on Packet Reception Ratio (PRR), and PRR is computed from the same data trace that has been used to measure the burstiness parameters. We explain how IM is computed by using Figure 7.9. Assume that link L_i spans from node N_{i1} to node N_{i2} and link L_j spans from node N_{j1} to node N_{j2}. We compute PRR of every link and then set $IM(i,j)$ to 1 if any of the links L_1, L_2, L_3 or L_4 have a PRR greater than PRR_t, a threshold. We set PRR_t to 0.3 in our experiment. This is a conservative model that hurts the latency bound, but improves the miss ratio.

7.6.4 Effect of $Bmax$

In this section, we describe the effect of $Bmax$ on end-to-end deadline miss ratio and latency bound. In our problem definition described in Section 7.3 the streams did not have any deadline associated with them and we define the deadline of the streams to be their generated latency bound. We evaluate it on the testbed. We compute the burstiness parameters $Bmax$,$B'min$ and the interference matrix IM of the actual testbed network exactly the way specified in Sections 7.6.2 and 7.6.3. To ensure packet transmission experiences the same burst-behavior, at this experiment we maintain the same packet transmission rate of around 200 packets per second with a zero inter packet interval that we use to measure link burstiness as described in Section 7.6.2.

To disable the effect of $B'min$ we select $B'min$ to 1 for all links. We define a multiplying factor K to demonstrate a trade off between latency bound and end-to-end deadline miss ratio and instead of allocating $Bmax + 1$ time slots per link, we are allocating $Bmax \times K + 1$ time slots for different values of K ranging from 0 to 2.

We run the experiment by considering two cases. In case 1, we use the whole testbed for evaluation. This case represents an actual deployed system. But in case 2, we assume that we don't have the top 25% links and we are forced to use some non-stationary links. This case represents another system where links are not as good. The criteria for discarding the top 25% links is $Bmax$ that assumes that the smaller the $Bmax$, the better the link is. Since we are not using the top 25% of the links in case 2, the workloads for these two cases are different. But for each case, for each value of K, we generate 10 different workloads and the average values are plotted in Figure 7.10.

To generate a workload, we randomly select 10 pairs of nodes as sources and destinations to generate 10 random streams, and choose the route of those streams as the shortest path from the source node to the destination node having minimum sum of $Bmax \times K + 1$ time slots. The starting time for every stream is set to its 1st time slot and period is randomly selected from a pool of even numbers up to 800 time slots for case 1 and 6000 time slots for case 2. If we consider a packet transmission rate of 200 packets per second, and allocate 5 msec per time slot, then the generated latency bound at K=1 is $51 \times 5 = 255$ msec for case 1, which is practical for the implementation of control loops in factory automation.

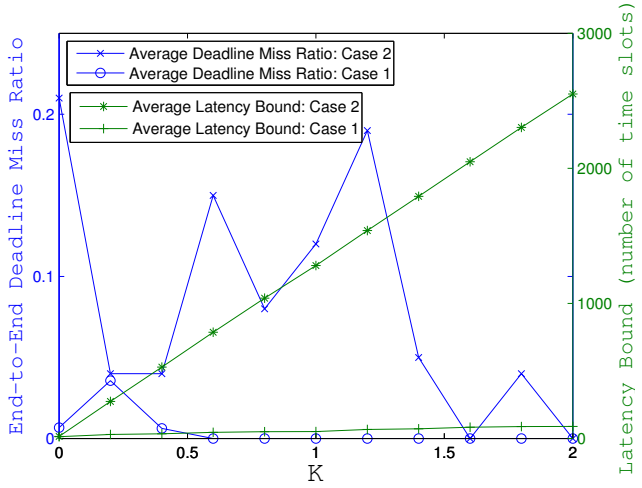


Figure 7.10: Effect of $Bmax$ on End-to-end Deadline Miss Ratio and Latency Bound

To time synchronize the nodes, we use a RBS style synchronization technique [128]. Since nodes may experience clock drift, to keep the nodes synchronized over time we need to allocate time slots for periodic broadcasting of the time-synchronization message.

Figure 7.10 shows the effect of $Bmax$ on latency bound. From the figure we observe that as K increases, the average latency bound increases linearly for both cases. The reason is, as K increases, $Bmax$ of all the links increases linearly and since $B'min$ is one for all the links, there is no overlapping of time slots to reduce latency bound.

Figure 7.10 also shows the effect of $Bmax$ on end-to-end deadline miss ratio in the testbed. For case 1, with the full testbed, we observe that although there are some fluctuations in the end-to-end deadline miss ratio for different values of K up to 0.6, the end-to-end deadline miss ratio becomes 0 after $K = 0.6$. This implies that when there are many good links in a network, our solution obtains 100% packet delivery within the latency bounds. Surprisingly, we observe that even before $K = 1.0$. Note that our generated latency bound (at K = 1) is within 14.2% of the minimum latency (at $K = 0.6$) for which we observe zero end-to-end deadline miss ratio. So, the generated latency bound is relatively tight. Also, if we allocate $0.6 * Bmax + 1$ time slots instead of $Bmax + 1$ time slots, we can save 12.4 % of average latency and can still make all the deadlines. We may not want to set $K < 1$ for hard real time applications, but it can be very useful to control the trade off between end-to-end deadline miss ratio and latency bound for soft real time applications.

For case 2, with the top 25% links removed, we observe a different result. Here, missing of deadlines continues beyond $K \geq 1$ even though we are allocating $Bmax \times K + 1$ time slots. The reason behind this is, links having high $Bmax$ are typically susceptible to the changes in the physical environment as shown in Figure 7.8 and to accurately characterize these links, empirical data should be collected over more than 21 days. Since are choosing least-burst-route for packet transmission, in case 1, we have sufficient number of links having low $Bmax$ for packet transmission while in case 2, we are forced to choose some links having high $Bmax$, which tend to be Epsilon Stationary links

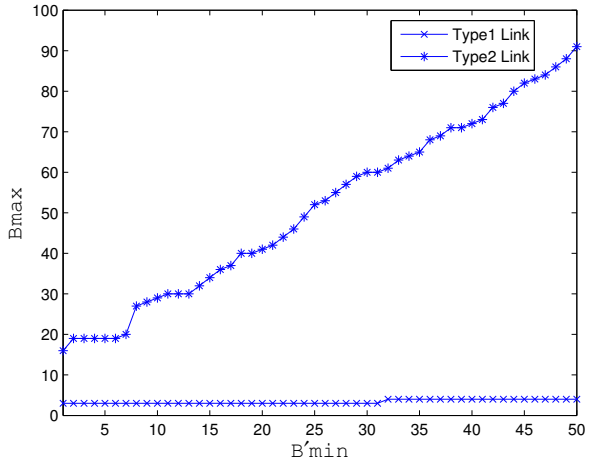


Figure 7.11: Variation of $Bmax$ for Different $B'min$

and Non-Stationary links. We can differentiate the links based on 21 days of link characterization as shown in Figure 7.6, but Epsilon Stationary links having high $Bmax$ require more than 21 days for a complete characterization of the environmental dynamics that affect them. The burst size may become bigger than the measured one no matter how long the empirical characterization is. In this case, we can address the problem with two different approaches: packet recovery and link adaptation as discussed in Section 7.7.2.

7.6.5 Effect of $B'min$

In this section we discuss the effect of $B'min$ on the latency bound. For a particular stream, its latency bound will either increase of remain same if $B'min$ is increased for every link that the stream goes through. The effect depends on the quality of the link. For high quality links, either stationary or asymptote stationary, the response to the increase of $B'min$ is of two types. So, we call these links as Type1 and Type2 links. The response of the two types of links to different values of $B'min$ is shown in Figure 7.11.

In a Type1 link, as we see from Figure 7.11, $Bmax$ increases very slowly with the increase of $B'min$. As a result, the latency bound remains the same for some time and increases very slowly for the increase of $B'min$. In the case of a Type2 link, $Bmax$ increases rapidly with the increase of $B'min$ and that's why the latency bound for a particular stream also increases rapidly. The reason behind this behavior depends on the link quality. After transmitting 3,600,000 packets over every link, we compute $Bmax$ for different values of $B'min$ by using algorithm 2. We observe that Type1 links are so good that even if we increase $B'min$, $Bmax$ remains almost the same and increases very slowly with keeping at least $B'min$ number of good slots for packet transmission in every possible window of size $Bmax$ + $B'min$. In the case of Type2 links, as $B'min$ increases, to keep at least $B'min$ number of good slots in every possible window of size $Bmax$ + $B'min$, we need to increase the window size by increasing $Bmax$ rapidly.

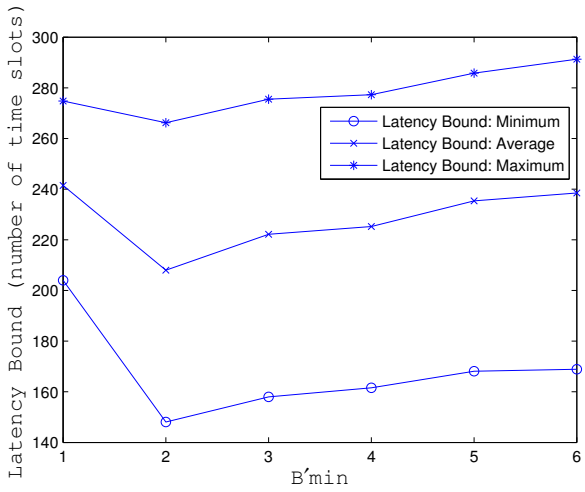


Figure 7.12: Effect of $B'min$ on Latency Bound

However, for a set of streams, the average latency bound may increase, decrease or remain same as a result of the increase of $B'min$ of every link depending on quality of the links, either Type1 or Type2, and spatio-temporal overlapping of the streams. If a large number of streams pass through a dense network with some spatial and temporal overlapping in transmission, we observe a decrease in average latency as $B'min$ increases, but after all the overlapping advantage is exploited, the average latency bound starts to rise again. We observe a similar result with 10 randomly generated streams for the testbed for which average latency bound decreases up to $B'min = 2$ and then it starts to rise again as is shown in Figure 7.12. We run the experiment 5 times for each value of $B'min$ and the minimum and maximum values of the latency bound of 5 runs are also plotted in the figure. It clearly indicates that we can minimize the average latency bound of a set of streams by an intelligent selection of $B'min$ of the links.

7.7 Discussion

In this section, we discuss the energy characteristics of the scheduling algorithm, specify two strategies to deal with a change of burst behavior, and suggest a few techniques to improve the scheduler.

7.7.1 Energy Characteristics

Although the scheduler allocates redundant time slots for packet transmission, since we use least-burst-routing, most of the selected links are very good and hence single packet transmission suffices in most of the cases. From both the transmitter and receiver's point of view, the radio needs to be only used for single packet transmission time in most of the cases, which is optimal. In the case of packet loss, the transmitter and receiver stay on until the packet is received. This energy consumption can only be beaten on the transmitter side by approaches such as [93], where the transmitter

turns off after a packet loss. However, that approach pays in terms of longer receiver wake times. Therefore, we expect our approach to have similar overall energy characteristics to [93].

7.7.2 Change of Burst Behavior

The burst-behavior of the wireless links may change due to a change in the physical environment, e.g. node failure, node replacement, or unexpected obstacles. To address this problem we propose two different approaches: packet recovery and link adaptation.

In a packet recovery approach, every node has a queue to hold the un-transmitted packet. The packet can be transmitted later at some other period if there are extra time slots left for packet transmission after all the packets of that period are transmitted successfully. In this way, the packet reaches the destination although it missed its deadline.

In the link adaptation approach, every node keeps a record about the links when it fails to transmit a packet over those links trying all the allocated time slots. After a specific interval, all the nodes report to some base station about the links over which packet transmission failed. The base station reschedules the packet transmission by doubling the allocated time slots at those links. This approach can be used to dynamically shrink and expand the latency bound based on network dynamics. We consider the evaluation of these two approaches as future work.

7.7.3 Scheduler Improvement

In this section, we suggest a few techniques that will improve the performance of the scheduler, which are considered for the future work. Instead of back-to-back retransmission, the scheduler can be smart enough to defer retransmissions a little, or retransmit on another channel, or use an alternate route, or even employ a combination of these based on the link characteristics. Also, in this work, we set the same $B\prime min$ on all the links. To minimize latency bound by a better selection of $B\prime min$ we need to try different values of $B\prime min$ starting from 1 until the latency bound starts to rise as shown in Figure 7.12. To set optimal values of $B\prime min$ to minimize the latency bound, where different links can have different values of $B\prime min$, we need to use a more sophisticated algorithm e.g. simulated annealing within the scheduler.

7.8 Summary

Some smart home apps may have dependencies on real-time constraints. They require reliable and on time packet delivery over a wireless network, which is very difficult to accomplish without any support from the underlying platform, DepSys. “Real-Time Delivery Scheduler” module in DepSys (Figure 7.1) provides a service to address such dependencies. In this chapter, we present how “Real-Time Delivery Scheduler” generates a schedule for packet transmission that produces an upper bound on the latencies of the streams. In order to do that, the solution addresses

dependencies on two physical properties of the communication environment, link burstiness and interference. We expect that this technique will help enable the use of wireless sensor and actuator networks in real-time applications in smart homes and in elsewhere.

Chapter 8

Conclusions

This thesis addresses several key challenges for integrating the Cyber-Physical Systems (CPSs) in a smart home and significantly advances the state of the art. In this final chapter, we describe the key contributions it makes, its limitations, and provide directions for future improvements.

8.1 Key Contributions

This thesis makes the following key contributions:

8.1.1 A Utility Sensing and Actuation Infrastructure for Smart Homes

At the core of this thesis, we present DepSys, a utility sensing and actuation infrastructure for smart homes that integrates many Cyber-Physical Systems (CPSs) of a smart home by treating each system as an app. DepSys provides a framework for the running of many smart home apps from many domains including energy, health, security, and entertainment. To do that, DepSys addresses a spectrum of dependencies including requirements, name, control, sensor reliability, real-time constraints, and human-in-the-loop dependencies. In addition, DepSys handles the case when app developers fail to specify dependencies. This thesis provides a detailed description of the comprehensive strategies that DepSys employs in order to address these dependency issues, which will lay the foundations for the integration of the future CPSs of smart homes and other CPS domains, e.g., industrial plants, where multiple systems need to share the underlying sensor and actuator utilities.

8.1.2 Novel Metadata For Actuator Level Conflict Detection and Resolution

The sharing of actuator utility across multiple simultaneously running apps in a smart home results in many systems-of-systems dependency problems. To address these control dependencies at the actuator level, in this thesis, we present

novel metadata: *effect*, *emphasis*, and *condition*. We are the first to demonstrate that dependency metadata (*effect*) that focuses on the effect on the environment enables us to detect conflicts across devices that isn't possible by monitoring actuations of individual devices as performed by the state of the art solutions. Also, our proposed novel metadata (*emphasis*) helps understanding the context and resolving actuator level conflicts more accurately that couldn't be achieved through existing priority based solutions. In addition, our proposed Semantic Aware Multilevel Equivalence Class based Policy (SAMECP) for resolving control conflicts of actuators reduces cognitive burden of the users and allows apps to be run in a more flexible way than the state of the art solution. By using 219 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 actuator level conflicts when multiple CPSs are integrated in a home setting and the significant ability of detecting and resolving such conflicts using these meta data.

8.1.3 Addressing Dependencies on Sensor Reliability Status

Apps that use sensors have dependencies on the reliability status of the sensors. DepSys runs a service to detect various types of realistic sensor failure in a home setting and notifies an app when a sensor that the app uses is detected as failed. We are the first to show that sensor-appliance behavioral patterns could be exploited to detect not only *fail-stop* failure, but *obstructed-view*, and *moved-location* failures that are common smart homes, very difficult to detect even if sensors are made highly reliable, and barely addressed to date. In this thesis, we present our novel failure detection solution, FailureSense that uses this insight and overcomes several limitations of the state of the art solutions, including it requires much less training effort, it is scalable in detecting multiple sensor failure even if they fail simultaneously, and it doesn't require sensor redundancy. By using 71 days of data from three real home deployments, we observe that FailureSense can detect *obstructed-view*, *moved-location* and *fail-stop* failures with 82.84%, 90.53%, and 86.87% precision, respectively, with an average of 88.81% recall. FailureSense is applicable to most of the common types of sensors found in real deployments thereby significantly improving the state of art.

8.1.4 Comprehensive and Personalized Human-in-the-loop Dependency Analysis

Multiple interventions from multiple simultaneously running human-in-the-loop apps raise complex dependency issues for controlling human physiological parameters. In this thesis, we present EyePhy, which is the first solution that directly addresses these issues. EyePhy runs as a service in the DepSys platform and uses a novel approach by employing a medically accepted physiological simulator that can model the complex interactions of the human physiology using over 7800 variables, and provides the most comprehensive dependency analysis involving human-in-the-loop known so far. It takes into account drug dosage and time gaps between the interventions, it reduces app developers efforts in specifying dependency metadata, and it offers personalized dependency analysis for the user. By using 12 human-in-the-loop

apps and a physiological simulator [73], we demonstrate the severity of conflicts across the interventions of the human-in-the-loop apps and the significant ability of EyePhy to detect these dependency issues.

8.1.5 Addressing Dependencies on Real-Time Constraints

Some apps that DepSys runs in a smart home may have dependencies on real-time constraints requiring reliable and on time packet delivery over wireless networks. DepSys offers a service to satisfy such dependencies. Based on significant empirical evidence of 21 days and over 3,600,000 packets transmission per link, we present a scheduling algorithm that produces latency bounds of the real-time periodic streams by addressing dependencies on two physical properties of the communication environment, link burstiness and interference. The solution is achieved through the definition of a new metric $Bmax$ that characterizes links by their maximum burst length, and by choosing a novel least-burst-route that minimizes the sum of worst case burst lengths over all links in the route. A testbed evaluation consisting of 48 nodes spread across a floor of a building shows that we obtain 100% reliable packet delivery within derived latency bounds. We also demonstrate how performance deteriorates and discuss its implications for wireless networks with insufficient high quality links.

8.2 Limitations and Future Improvements

First, security and privacy issues of residents are not addressed in this thesis. An attacker may compromise the platform, or an app, or a device to unlock the door while the residents are sleeping, or may access cameras in the home to obtain private photos. This is still an open problem and a promising direction for future work.

Second, DepSys does not address safety-critical issues in a home setting. For example, to resolve control dependencies at the actuators, DepSys uses the policy defined by the user. However, the residents may choose an erroneous policy that may jeopardize their health, e.g., assigning a higher priority to an energy management app over a health care app that controls the breathing machine. Also, the average median latency to detect various types of sensor failure by using FailureSense is 22.08 hours, which is not adequate for safety-critical applications, e.g., emergency health care and security apps.

Third, although our proposed metadata, *effect*, *emphasis*, and *condition* are powerful enough to detect conflicting actuations that occur within and across devices, these metadata only provide the basic dependency information of an app about using an actuator. We do not allow the app developers to specify complex dependency information using fuzzy logic, e.g., *App1* will use Light1 with 2% probability, or using complex temporal logic, e.g., *App1* will turn on the AC if the *eventual* temperature reaches 100 degree F, or using a combination of both, e.g., *App1* will turn on Light1 with

70% probability *as long as* Light2 is off. As a result, DepSys's installation time dependency checking at the actuator level is very conservative in that if the probability of conflict is very low, still it assumes that there will be conflicts.

Fourth, DepSys is completely agnostic about the application semantics of the apps. As a result, the conflict it detects may not be perceived as a conflict to the user or to the apps. For example, DepSys's human-in-the-loop dependency checker, EyePhy detects a conflict among interventions when it observes the opposite effect on a physiological parameter of interest. However, some conflicting interventions can be intentional and useful, e.g., e.g., to control self-consciousness, it may be acceptable to the residents to take caffeine with alcohol. But EyePhy can't determine it as it doesn't have the knowledge of the application semantics and the relevant contextual information. A promising future research direction is to determine the application semantics the app developers need to encode as app metadata, the language they need to use, and the enhancement it will provide to the DepSys's conflict detection and resolution capability.

Fifth, the Preference Learning Module of DepSys takes user input to determine app priorities, which is used in policies for resolving control conflicts at the sensor and actuator utilities. However, it does not use any machine learning algorithm to automatically learn the policies based on the lifestyles of the residents. Future smart home systems will benefit if a service can be built that automatically learns the preference of the residents by monitoring their lifestyles and necessary contextual information. It will reduce the cognitive burden of the users in specifying conflict resolution policies.

Finally, we use a handful of apps that are common in the literature but relatively simple at the time of designing and evaluating the dependency analysis techniques involving actuators and human-in-the-loop. As apps get more complicated, more complex dependency analysis may be needed, which may require additional metadata providing more contextual information.

Bibliography

[1] Timothy W. Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I. Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The hitchhiker's guide to successful residential sensing deployments. In *SenSys*, 2011.

[2] Add-in Model. `http://msdn.microsoft.com/en-us/library/bb384200(v=vs.110).aspx`.

[3] Control4 Home Automation and Control. `http://www.control4.com`.

[4] Home Automation Systems. HomeSeer. `http://www.homeseer.com`.

[5] M1 Security and Automation Controls. `http://www.elkproducts.com/m1_controls.html`.

[6] Leviton Online Store. `http://www.levitonproducts.com`.

[7] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A.J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *NSDI*, 2012.

[8] 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.

[9] Robert F. Dickerson, Eugenia I. Gorlin, and John A. Stankovic. Empath: a continuous remote emotional health monitoring system for depressive illness. In *WH*, 2011.

[10] D Cook and M Schmitter-Edgecombe. Assessing the quality of activities in a smart environment. *Methods of Information in Medicine*, 2009.

[11] Julie A. Kientz, Shwetak N. Patel, Brian Jones, Ed Price, Elizabeth D. Mynatt, and Gregory D. Abowd. The georgia tech aware home. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '08, pages 3675–3680, New York, NY, USA, 2008. ACM.

[12] Robert J. Orr and Gregory D. Abowd. The smart floor: A mechanism for natural user identification and tracking. In *CHI '00 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '00, pages 275–276, New York, NY, USA, 2000. ACM.

[13] Stephen S. Intille, Kent Larson, J. S. Beaudin, J. Nawyn, E. Munguia Tapia, and P. Kaushik. A living laboratory for the design and evaluation of ubiquitous computing technologies. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '05, pages 1941–1944, New York, NY, USA, 2005. ACM.

[14] David Malan, Thaddeus Fulford-jones, Matt Welsh, and Steve Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *In International Workshop on Wearable and Implantable Body Sensor Networks*, 2004.

[15] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. The gator tech smart house: a programmable pervasive space. *Computer*, 38(3):50–60, March 2005.

[16] Android. `http://www.android.com/`.

[17] iPhone. http://www.apple.com/iphone/.

[18] ADT. `http://www.adt.com`.

[19] Vivint. http://www.vivint.com.

[20] Honeywell. http://www.honeywellcity.com.

[21] Bosch. http://www.bosch.us/en/us/startpage_1/country-landingpage.php.

[22] Philips. http://www.usa.philips.com.

[23] SimpliSafe. http://simplisafe.com.

[24] Viper. http://www.viper.com/Home/.

[25] Alarm.com. http://www.alarm.com.

[26] Monitronics. http://www.monitronics.com.

[27] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[28] John Hatcliff, William Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *ICSE*, 2003.

[29] Matthew B Dwyer, John Hatcliff, et al. Bogor: an extensible and highly-modular software model checking framework. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 267–276. ACM, 2003.

[30] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1):161–183, 2005.

[31] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: a flexible and adaptive framework sensor networks. In *EWSN*, 2005.

[32] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *ICDCS*, 2005.

[33] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys*, 2003.

[34] Yang Yu, Loren J. Rittle, Vartika Bhandari, and Jason B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys*, 2006.

[35] L. Szumel, J. LeBrun, and J. D. Owens. Towards a mobile agent framework for sensor networks. In *EmNets*, 2005.

[36] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[37] Philip Levis, David Gay, and David Culler. Active sensor networks. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 343–356, Berkeley, CA, USA, 2005. USENIX Association.

[38] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[39] Gavin A. Campbell. Sensor network policy conflicts. In *IMAG Laboratory, University of Grenoble*, 2007.

[40] Stephan Reiff-marganiec and Kenneth J. Turner. Appel: the accent project policy environment/language, December 2005.

[41] Pascal A. Vicaire, Zhiheng Xie, Enamul Hoque, and John A. Stankovic. Physicalnet: A generic framework for managing and programming across pervasive computing networks. In *RTAS*, 2010.

[42] Pascal A. Vicaire, Enamul Hoque, Zhiheng Xie, and John A. Stankovic. Bundle: a group based programming abstraction for cyber physical systems. In *ICCPS*, 2010.

[43] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, March 1989.

[44] Stanislav Rost and Hari Balakrishnan. Memento: A health monitoring system for wireless sensor networks. In *SECON*, 2006.

[45] Bor-Rong Chen, Geoffrey Peterson, Geoff Mainland, and Matt Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *DCOSS*, 2008.

[46] Linnyer Ruiz, Jose Nogueira, and Antonio Loureiro. MANNA: A management architecture for wireless sensor networks. In *IEEE Communications Magazine*, February 2003.

[47] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *SenSys*, 2005.

[48] Kevin Ni, Nithya Ramanathan, Mohamed Nabil Hajj Chehade, Laura Balzano, Sheela Nair, Sadaf Zahedi, Eddie Kohler, Greg Pottie, Mark Hansen, and Mani Srivastava. Sensor network data fault types. In *ACM Trans. Sen. Netw.*, May 2009.

[49] Krasimira Kapitanova, Enamul Hoque, John A. Stankovic, Sang H. Son, and Kamin Whitehouse. Being SMART about failures: Assessing repairs in smart homes. In *UbiComp*, 2012.

[50] Saurabh Ganeriwal, Laura K. Balzano, and Mani B. Srivastava. Reputation-based framework for high integrity sensor networks. *ACM Trans. Sen. Netw.*, 2008.

[51] Shuo Guo, Ziguo Zhong, and Tian He. Find: faulty node detection for wireless sensor networks. In *SenSys*, 2009.

[52] George W Hart. Nonintrusive appliance load monitoring. In *Proceedings of the IEEE*, December 1992.

[53] Sidhant Gupta, Matthew S. Reynolds, and Shwetak N. Patel. Electrisense: single-point sensing using EMI for electrical event detection and classification in the home. In *UbiComp*, 2010.

[54] Shwetak N. Patel, Thomas Robertson, Julie A. Kientz, Matthew S. Reynolds, and Gregory D. Abowd. At the flick of a switch: detecting and classifying unique electrical events on the residential power line. In *UbiComp*, 2007.

[55] Younghun Kim, Thomas Schmid, Zainul M. Charbiwala, and Mani B. Srivastava. Viridiscope: design and implementation of a fine grained power monitoring system for homes. In *UbiComp*, 2009.

[56] Xiaofan Jiang, Stephen Dawson-Haggerty, Prabal Dutta, and David Culler. Design and implementation of a high-fidelity AC metering network. In *IPSN*, 2009.

[57] Sirajum Munir, John A. Stankovic, Chieh-Jan Mike Liang, and Shan Lin. Cyber physical system challenges for human-in-the-loop control. In *Feedback Computing*, 2013.

[58] David Arney, Miroslav Pajic, Julian M. Goldman, Insup Lee, Rahul Mangharam, and Oleg Sokolsky. Toward patient safety in closed-loop medical device systems. In *ICCPS*, 2010.

[59] Insup Lee, Oleg Sokolsky, Sanjian Chen, John Hatcliff, Eunkyoung Jee, BaekGyu Kim, Andrew King, Margaret Mullen-Fortino, Soojin Park, Alexander Roederer, and Krishna K. Venkatasubramanian. Challenges and research directions in medical cyber-physical systems. *Proceedings of the IEEE*, 2012.

[60] Anthony D. Wood and John A. Stankovic. Human in the loop: distributed data streams for immersive cyber-physical systems. *SIGBED Rev.*, 2008.

[61] Yufeng Xin, I. Baldine, J. Chase, T. Beyene, B. Parkhurst, and A. Chakrabortty. Virtual smart grid architecture and control framework. In *Smart Grid Communications (SmartGridComm)*, 2011.

[62] Xue Liu, Hui Ding, Kihwal Lee, Lui Sha, and Marco Caccamo. Feedback fault tolerance of real-time embedded systems: issues and possible solutions. *SIGBED Rev.*, April 2006.

[63] Sirajum Munir, Jonh A. Stankovic, Chieh jan Mike Liang, and Shan Lin. Reducing energy waste for computers by human-in-the-loop control. *IEEE Transactions on Emerging Topics in Computing*, to appear.

[64] Jiakang Lu, Tamim Sookoor, Vijay Srinivasan, Ge Gao, Brian Holben, John Stankovic, Eric Field, and Kamin Whitehouse. The smart thermostat: using occupancy sensors to save energy in homes. In *SenSys*, 2010.

[65] Yuvraj Agarwal, Bharathan Balaji, Seemanta Dutta, Rajesh K. Gupta, and Thomas Weng. Duty-cycling buildings aggressively: The next frontier in HVAC control. In *IPSN*, 2011.

[66] Yong Fu, Nicholas Kottenstette, Yingming Chen, Chenyang Lu, Xenofon D. Koutsoukos, and Hongan Wang. Feedback thermal control for real-time systems. In *RTAS*, 2010.

[67] Dae-Jin Kim and Aman Behal. Human-in-the-loop control of an assistive robotic arm in unstructured environments for spinal cord injured users. In *HRI*, 2009.

[68] William T Riley, Daniel E Rivera, Audie A Atienza, Wendy Nilsen, Susannah M Allison, and Robin Mermelstein. Health behavior models in the age of mobile interventions: are our theories up to the task? *Translational Behavioral Medicine*, 1(1):53–71, 2011.

[69] Susan Michie, Maartje M van Stralen, and Robert West. The behaviour change wheel: a new method for characterising and designing behaviour change interventions. *Implementation Science*, 6(42), 2011.

[70] Charles Abraham and Susan Michie. A taxonomy of behavior change techniques used in interventions. *Health Psychology*, 27(3):379–387, May 2008.

[71] Susan Michie, Marie Johnston, Jill Francis, Wendy Hardeman, and Martin Eccles. From theory to intervention: Mapping theoretically derived behavioural determinants to behaviour change techniques. *Applied Psychology*, 57(4):660–680, 2008.

[72] Susan Michie and Andrew Prestwich. Are interventions theory-based? development of a theory coding scheme. *Health Psychology*, 29(1):1–8, 2010.

[73] Robert Hester, Alison Brown, Leland Husband, Radu Iliescu, William Andrew Pruett, Richard L Summers, and Thomas Coleman. Hummod: A modeling environment for the simulation of integrative human physiology. *Frontiers in Physiology*, 2(12), 2011.

[74] M. Rupnik, F. Runovc, D. Sket, and M. Korda. Cardiovascular physiology: simulation of steady state and transient phenomena by using the equivalent electronic circuit. *Computer Methods and Programs in Biomedicine*, 67(1):1 – 12, 2002.

[75] A. Das, Z. Gao, P. P. Menon, J. G. Hardman, and D. G. Bates. A systems engineering approach to validation of a pulmonary physiology simulator for clinical applications. *Journal of The Royal Society Interface*, 8(54):44–55, 2011.

[76] M. Kordas, S. Leonardis, and J. Trontelj. An electrical model of blood circulation. *Medical and biological engineering*, 6(4):449–451, 1968.

[77] Daniel A Beard. Modeling of oxygen transport and cellular energetics explains observations on in vivo cardiac energy metabolism. *PLoS Comput Biol*, 2(9):e107, 09 2006.

[78] TimothyW. Secomb, Beata Styp-Rekowska, and AxelR. Pries. Two-dimensional simulation of red blood cell deformation and lateral migration in microvessels. *Annals of Biomedical Engineering*, 35(5):755–765, 2007.

[79] J G Hardman, N M Bedforth, A B Ahmed, R P Mahajan, and A R Aitkenhead. A physiology simulator: validation of its respiratory components and its ability to predict the patient's response to changes in mechanical ventilation. *British Journal of Anaesthesia*, 81(3):327–32, 1998.

[80] A C Guyton, T G Coleman, and H J Granger. Circulation: Overall regulation. *Annual Review of Physiology*, 34(1):13–44, 1972. PMID: 4334846.

[81] Thomas G. Coleman and James E. Randall. Human—a comprehensive physiological model. Technical report, The Physiologist, 1983.

[82] Sean R. Abram, Benjamin L. Hodnett, Richard L. Summers, Thomas G. Coleman, and Robert L. Hester. Quantitative circulatory physiology: an integrative mathematical model of human physiology for medical education. *Advances in Physiology Education*, 31(2):202–210, 2007.

[83] A. Cerpa, J. L. Wong, M. Potkonjak, and D. Estrin. Temporal properties of low power wireless links: Modeling and implications on multi-hop routing. In *MobiHoc '05*.

[84] D. Lymberopoulos, Q. Lindsey, and A. Savvides. An empirical analysis of radio signal strength variability in ieee 802.15.4 networks using monopole antennas. In *ENALAB Technical Report 050501, EWSN 2006*.

[85] S. Lin, G. Zhou, K. Whitehouse, Y. Wu, J. A. Stankovic, and Tian He. Towards stable network performance for wireless sensor networks. In *IEEE RTSS '09*.

[86] V. Raghunathan, S. Ganeriwal, C. Schurgers, and M. B. Srivastava. Energy efficient wireless packet scheduling and fair queuing. In *ACM Transactions on Embedded Computing Systems*, 2004.

[87] S. Lin, J. Zhang, G. Zhou, L. Gu, T. He, and J. A. Stankovic. Atpc: Adaptive transmission power control for wireless sensor networks. In *ACM SenSys '06*.

[88] G. Hackmann, O. Chipara, and C. Lu. Robust topology control for indoor wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, 2008.

[89] A. Willig. How to exploit spatial diversity in wireless industrial networks. In *IFAC Annual Reviews in Control*, 2008.

[90] P. Agrawal and N. Patwari. Correlated link shadow fading in multi-hop wireless networks. In *Tech Report arXiv:0804.2708v2*, 2008.

[91] G. Zhou, T. He, S. Krishnamurthy, and J. A. Stankovic. Impact of radio irregularity on wireless sensor networks. In *ACM MOBISYS 2004*.

[92] I. Hou, V. Borkar, and P. R. Kumar. A theory of qos for wireless. In *Infocom 2009*.

[93] K. Srinivasan, M. A. Kazandjieva, S. Agarwal, and P. Levis. The β-factor: Measuring wireless link burstiness. In *SenSys '08*.

[94] Wirelesshart overview. `http://en.hartcomm.org/hcp/tech/aboutprotocol/aboutprotocol_what.html`.

[95] J. Broch, D. A. Maltz, D. B. Johnson, Y. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Mobicom '98*.

[96] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, 1996.

[97] C. Lu, B. Blum, T. Abdelzaher, J. Stankovic, , and T. He. Rap: a real-time communication architecture for large-scale wireless sensor networks. In *RTAS '02*.

[98] T. He, J. A. Stankovic, C. Lu, and T. F. Abdelzaher. Speed: A stateless protocol for real-time communication in sensor networks. In *ICDCS '03*.

[99] T. L. Crenshaw, S. Hoke, A. Tirumala, and M. Caccamo. Robust implicit edf: A wireless mac protocol for collaborative real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[100] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 1973.

[101] H. Li, P. Shenoy, and K. Ramamritham. Scheduling messages with deadlines in multi-hop real-time sensor networks. In *RTAS '05*.

[102] `http://en.wikipedia.org/wiki/Google_Play`.

[103] `http://en.wikipedia.org/wiki/App_Store_(iOS)`.

[104] `http://www.avatargeneration.com/2013/09/how-many-people-use-mobile-apps-around-the-world/`.

[105] Vaughn Bradshaw. *The Building Environment: Active and Passive Control Systems*. John Wiley & Sons, Inc., River Street, NJ, USA, 2006.

[106] Apple app store. `http://www.apple.com/osx/apps/app-store.html`.

[107] Google play. `http://play.google.com/store?hl=en`.

[108] `http://velositor.com/2012/02/27/the-average-iphone-user-has-44-apps-on-their-device-versus-only-32-apps-for-android-smartphone-users/`.

[109] TED. The Energy Detective. `http://www.theenergydetective.com/`.

[110] eMonitor. `http://www.powerhousedynamics.com/`.

[111] IntelligentUtility report. http://www.intelligentutility.com/magazi ne/article/253959/6027-million-installed-smart-meters-globally- 2016.

[112] ON World report. http://www.onworld.com/smartmeterset/HANrpt.html.

[113] Rayoung Yang and Mark W. Newman. Learning from a learning thermostat: lessons for intelligent systems for the home. In *UbiComp*, 2013.

[114] Beth Logan, Jennifer Healey, Matthai Philipose, Emmanuel Munguia Tapia, and Stephen Intille. A long-term evaluation of sensing modalities for activity recognition. In *UbiComp*, 2007.

[115] D Cook and M Schmitter-Edgecombe. Assessing the quality of activities in a smart environment. *Methods of Information in Medicine*, 2009.

[116] Tim van Kasteren, Athanasios Noulas, Gwenn Englebienne, and Ben Kröse. Accurate activity recognition in a home setting. In *UbiComp*, 2008.

[117] Vijay Srinivasan, John Stankovic, and Kamin Whitehouse. Watersense: Water flow disaggregation using motion sensors. In *BuildSys*, 2011.

[118] http://lifescientist.com.au/content/biotechnology/article/the-rise-of-smartphone-health-and-medical-apps-1072193834.

[119] http://reference.medscape.com/drug-interactionchecker.

[120] http://www.drugs.com/drug_interactions.php.

[121] http://www.nlm.nih.gov/medlineplus/druginformation.html.

[122] http://www.nlm.nih.gov/medlineplus/ency/article/002341.htm.

[123] http://www.informationweek.com/regulations/lawmakers-try-to-sharpen-fda-focus-on-healthcare-apps/d/d-id/1112095?

[124] Andrew P Demidowich, Kevin Lu, Ronald Tamler, and Zachary Bloomgarden. An evaluation of diabetes self-management applications for android smartphones. *J Telemed Telecar*, 18(4), 2012.

[125] M. Franceschinis, M. A. Spirito, R. Tomasi, G. Ossini, and M. Pidala. Using wsn technology for industrial monitoring: A real case. In *SENSORCOMM '08*.

[126] T. Rusak and P. Levis. Burstiness and scaling in low power wireless simulation. In *MobiCom'08*.

[127] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, 1997.

[128] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *(OSDI 2002)*.