CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Mobile-based protocol-agnostic control over the smart home

Master's thesis in Computer Science – Algorithms, Languages and Logic

JOHANNES KEINESTAM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

MASTER'S THESIS 2016

Mobile-based protocol-agnostic control over the smart home

JOHANNES KEINESTAM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Mobile-based protocol-agnostic control over the smart home
JOHANNES KEINESTAM

Supervisor: Carlo A. Furia, Department of Computer Science and Engineering
Advisor: Eric Calissendorff, Plejd AB
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden 2016

JOHANNES KEINESTAM
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

This Master's thesis describes the development of an interoperability model for allowing control of heterogeneous smart home devices through a single smart home control system. The resulting model is implemented in a mobile application intending to provide reliable and performant device communication.

The developed interoperability model allows support for devices to be added to a system implementing the model by writing device drivers which describe a physical device by its metadata and functionality. The functionality of a device is modeled as a list of controls which map the functionality provided by the device to functionality understood by the interoperability model. A control used by a driver must fulfill the contract of one of eight pre-defined controls, thus allowing them to be appropriately rendered in a graphical user interface. The mobile application implementation of the interoperability model uses dynamic code loading to load device drivers written in Lua, for which drivers supporting smart lights from Philips and Plejd are implemented and tested.

Evaluation by applying the abstractions of the model to various types of devices finds the model to be highly applicable. However, it is noted that the interoperability model does not support devices which communicate large quantities of binary data such as cameras or media streaming systems. Testing of the implementation further shows that device drivers perform well in a high traffic test environment. The interoperability model is deemed by the author to be relevant to the smart home field through its applicability and ease-of-use, but it is noted that the model should be further tested in real-world environments.

Keywords: *interoperability, internet of things, device communication, smart home*

Acknowledgements

The author wishes to extend his sincere gratitude to all the people who have provided feedback during the work on this thesis. The advisor, Erik Calissendorff, the academic supervisor, Carlo A. Furia, and the examiner, Alejandro Russo, have in particular been of great help during the project. Magnus Ekberg, Babak Esfahani and the rest of the employees of Plejd have additionally provided support and help in technical matters. Furthermore, the author wishes to thank all the people who have helped through peer review and thesis opposition for valuable help in shaping this report.

Table of Contents

Chapter 1

Introduction

This chapter will introduce a Master's thesis intending to present an interoperability model for smart home devices, developed in conjunction with Plejd AB.

1.1 Background

The smart home is grounded in the idea of Building Automation, which allows centralized control and automation of building environments, such as heating, lighting, and security. The concept of the smart home extends building automation with the idea of controlling household goods like home entertainment systems and household appliances. Such automation can be motivated by energy savings and environmental considerations which are impacted by optimizing the usage of heating and lighting. Further motivations include comfort and time-savings by automating menial tasks.

With the advancement in communication technology and the ubiquitous nature of small, connected devices, sometimes dubbed the Internet of Things, the smart home is an increasingly viable concept. Manufacturers such as Samsung, Nest Labs, and Philips have launched smart home appliances, each within their own ecosystem. As the focus of innovation has largely been on single devices with little focus on the bigger picture, such ecosystems offer little interoperability with competitor's products, and a consumer's choice of one manufacturer's system may limit them in the smart products they may be able to use in their smart home.

1.2 Aim

This project concerns the development of a model which can facilitate interoperability among smart home devices, and implementation of the model in smart home control units running on mobile devices. The control units will allow control of smart home devices from

vendors using different communication protocols. An interface will be provided for communication with devices, allowing developers to add support for various heterogeneous protocols. The project thus includes two different but related parts: an interoperability model and an application which will apply this model. As such, the aims can be further examined for each part.

The interoperability model will extract common elements of relevant communication protocols for the sake of providing an abstraction aiding third party development of device support. The aim is that these abstractions should allow developers to add support for protocols without knowing details of the model or the application.

The developed interoperability model will be applied in practice in a pre-existing iOS and Android application which currently allows control of Plejd light mesh devices. The application should be extended with a module applying the model as a layer in its communication stack. From a user perspective, the aim is to create a seamless user experience where one can add devices to one's smart home without finding and installing new software.

1.3 Problem formulation

The thesis includes development of two different parts, which come with their own challenges.

The interoperability model is central to the thesis, as it will give a clear picture of how smart home protocols may be structured, classified, and allowed to intercommunicate. It will present numerous challenges in finding clear solutions that will work generally, yet provide good performance for various smart home environments. The research question of the interoperability model to be answered is:

- Which classifications and abstractions are important when facilitating inter-protocol communication of smart home devices?

The application implementation presents challenges from an engineering perspective, where it will consist of reasoning about how to produce a smooth user experience. This requires allowing protocol support plugins being loaded without interruption (possibly automatically) and immediately letting the control unit communicate with smart home devices, as well as developing a reliable communication stack which can handle exceptional software or network behavior. The research questions to be answered for this section are:

- How can plugins for protocol support be loaded on the fly, or even automatically?
- How to allow reliable and performant communication with heterogeneous devices from a mobile device?

1.4 Related work

Manufacturers of smart home devices (among them Samsung, Nest Labs, and Philips), while providing their own infrastructures, have open sourced their protocols so that interested developers can control their devices.

Interoperability between units has previously been investigated. Baumeister & Fuchs (2015) described considerations for communication between devices in the smart home and summarized attempted standardization of interoperability among appliances, among them an on-going standardization effort by the European Committee for Electrotechnical Standardization.

Perumal et al. (2008) investigated the possibility of employing Web Service technology using Simple Object Access Protocol (SOAP). An interoperability tier acts as the middleware, consulting a database of known services when a SOAP message is received, and gives a response to another device according to defined rules. The solution was found to be performant in regards to response time in an evaluation of interoperability between two subsystems communicating over Ethernet. An extended implementation using an Event-Condition-Action mechanism for specifying rules was presented in Leong et al. (2009).

Capitanelli et al. (2014) described a methodology for classifying devices and defining rules for their communication using a set of predefined functionality, device, and information categories. A simulated smart home environment case study is evaluated from a user centric perspective, finding the proposed model flexible and adaptable to multiple smart home systems. Maestre & Camacho (2009) presented a smart home system based on the UPnP standard. The system running on Windows computers supports interoperability with devices which were using UPnP, ZigBee, and X-10.

Most of the described papers present largely theoretical results in a smaller area of smart home technology. They are not focused on presenting a complete implementation of a method for classifying devices, as well as device communication and plugin protocol support. The interoperability model presented by Perumal et al. (2008) and Leong et al. (2009) considered systems which support communication with SOAP. Furthermore, the implementation is given a secondary role, and the algorithmic and software design considerations are outside their scope. In contrast, this project aims to contribute a widely applicable model supporting heterogeneous communication protocols which can be applied in smart home control systems requiring little knowledge of the types of devices it is being used for, yet present and control the devices in a suitable way.

1.5 Scope

As the project concerns communication between heterogeneous devices, the scope can easily grow very large. The project is made feasible in the scheduled 20 week time frame by a set of limitations, which follow.

- The application module, while highly extensible, should support a few given device protocols. For the scope of this thesis, the focus will be on Philips Hue and Plejd lights.
- The application will be developed for mobile target platforms, and will thus not be able to support some communication protocols which the platform cannot support. The communication channels to be supported by the implementation are HTTP-based communication and Bluetooth Low Energy.

Chapter 2

Method

This section will describe the methodology which was used in the thesis work in three different phases: state of the art research, interoperability model development, and application module implementation.

2.1 State of the Art

The first phase of the project involved research of the state of the art. A collection of protocols used by existing smart home devices was studied, and their capabilities and technologies were noted. Research into interoperability between smart home devices and research into general protocol interoperability was collected from scientific as well as industry sources.

A summarization of the state of the art in smart home interoperability was contributed in Chapter 3, detailing a few select interoperability concepts and discussing their relevance to the interoperability model developed in this thesis. This summarization additionally served as a vital basis for the development of the interoperability model.

2.2 Interoperability model

The interoperability model was developed with respect to the findings of the state of the art portion, proving vital for presenting a good solution while taking advantage of the state of the art. The process of developing the model is described in Chapter 4.

The model utilizes several concepts for providing interoperability. In particular, a taxonomy for categorizing smart home devices was developed. This taxonomy is meant to be applied to any smart home device for which support is to be added, and thus reflects aspects such as the device's purpose, communication ability, discovery process and protocol. This categorization allows the interoperability model to dynamically support heterogeneous devices.

2.3 Application

To evaluate the interoperability model, it was applied as a module in an iOS/Android application as described in Chapter 5. The pre-existing application was extended to use the taxonomy and provided utilities of the model as a basis for communicating with concrete devices. The software architecture of the application module includes a set of submodules:

- Handling of communication via IP and Bluetooth Low Energy.
- Handling of discovery via IP and Bluetooth Low Energy.
- Dynamic loading of device support plugins.
- Describing and abstracting actions (read/write) that devices support.

The module was developed in Xamarin, which is a C#-based cross-platform solution for application development targeting mobile operating systems. The development was conducted using a semi-agile workflow, where the requirements specified were iteratively implemented and tested. As additional requirements were discovered throughout the development process, they were added to the requirement list and implemented in order of priority.

2.4 Evaluation

The encompassing and abstract nature of the model developed as part of this thesis made it difficult to formally evaluate. As such, in deciding the evaluation methodology numerous options were considered. The decided methodology involved theoretically applying the model to various smart home devices, and evaluating its applicability to heterogeneous devices from the results, as seen in Section 4.2.2. Additionally, the success of the implementation of the application module and the perceived ease of implementing plugins for device support is intended to provide an evaluation of the aim of allowing developers to easily add support for devices to a system which applies the interoperability model.

The application module was evaluated in Section 5.2.2 by testing its performance in real world environments where multiple devices and protocols were used. The performance of the solution was measured by response times and throughput of data, thus evaluating the possibility of controlling multiple devices simultaneously. These tests were conducted in a real environment in a large of a degree as possible, but simulations were employed when that was considered infeasible.

Chapter 3

State of the Art

This chapter will summarize the state of the art in research relating to interoperability in the smart home, for the purpose of informing and justifying choices made in the interoperability model presented in the next chapter.

Through research, a few common areas of study relating to interoperability have been identified: programming abstractions, device taxonomy, and common data models. Chosen key research in these areas will be presented and their relation to the interoperability model to be developed will be discussed. Furthermore, Wireless Sensor Networks as they relate to the smart home will be discussed.

3.1 Wireless Sensor Networks

In a Wireless Sensor Network (*WSN*), autonomous sensors are programmed to redirect data through the sensor network via a wireless multi-hop process to and from a central gateway. Sensors are a natural part of the smart home environment, as they may provide data such as ambient temperature, whether doors or windows are closed, and energy usage. As sensors are small and cheap, WSNs may prove to be an ubiquitous part of the smart home, and are thus relevant to the topic of this paper.

Wireless Sensor Networks is an active field of research and can be seen as related to smart home device interoperability. The research often deals with concepts such as abstraction of sensor functionality, their data model as well as message translation. Mottola and Picco (2011) present an overview of research in the area, where a taxonomy for programming abstractions is used to categorize a collection of available abstractions for WSNs. While there is a difference between simple sensors and the more feature-rich smart home devices which is the focus of the interoperability model to be developed, wireless sensor networks are nonetheless a relevant research area.

3.2 Programming abstractions

Providing programming abstractions can facilitate the programming of sensors or other devices in multiple ways. In the context of WSNs, programming abstractions can allow programmers to write a single program for the entire network which a compiler can then separate into discrete tasks that are distributed to the individual sensors. These abstractions can allow for implicitly addressing sensors, aggregating sensors depending on their values or their physical location, and sharing data among sensors, among other things. These features can greatly facilitate development, which is relevant to the application module to be developed.

Mottola and Picco (2011) present a number of different abstractions, and categorizes them according to their communication scope, addressing and awareness, computation scope, data access model and programming paradigm. Hadim and Mohamed (2006) further describe programming abstractions as separate from programming support, where the former refers to the way a sensor network is presented to the programmer while the latter concerns providing services, tools and runtime mechanisms to the programmer. A few programming abstractions from a variety of sources that were found interesting will be presented here.

Logical Neighborhoods is a programming abstraction which provides a declarative language called *Spidey* for specifying neighborhoods templates and node templates describing the attributes of a node (Mottola and Picco, 2006). The (logical) neighborhoods serve as the foundation of the provided abstraction, essentially allowing the programmer to collect nodes into a ”neighborhood” depending on the values of their attributes. These neighborhood templates can be instantiated with parameter values, as can be seen in Listing 3.1 where a neighborhood of temperature sensors reading a temperature over 100 degrees is created.

Listing 3.1: A Spidey code snippet showing the definition of a neighborhood, adapted from Mottola and Picco (2006)

```
neighborhood template HighTempSens(threshold)
    with Function = "sensor" and
         Type = "temperature" and
         Reading > threshold

create neighborhood hts100
    from HighTempSens(threshold : 100)
    max hops 2
    credits 30
```

Ravel provides a Model-View-Controller (MVC) programming abstraction for the sensor, gateway and cloud-hosted application/storage tiers (Riliskis et al., 2015). This allows the programmer to write a single MVC application which is then compiled to platform-specific

code across all three tiers. Facilitating this goal is the concept of *spaces*, essentially a set of rules for translating from the Ravel-defined code (in Riliskis et al. (2015), Python-based) to the specific code of each tier's platform. Spaces also allow the compiled code to leverage platform-appropriate data structures, as well as allowing the different tiers to store appropriate amounts of the sensed data.

The functionality of Ravel is exemplified in a water-saving application which uses temperature and flow sensors at water pipes. The sensed data is synchronized by Ravel to an Android smartphone serving as a gateway, which then synchronizes the data to a Django web application running in the cloud. Ravel allows the sensor to only store the current data, while the Android application can store a larger amount to provide e.g. daily averages of water consumption, and the cloud service can store the entire water consumption history in detail. These sort of abstractions provide a very powerful tool for the programmer, allowing them to consider the whole system as an application instead of as a set of devices.

In conclusion, while programming abstractions which are designed for WSNs are interesting, they are not entirely applicable to the interoperability model of this thesis. This is largely due to WSNs requiring cooperative communication, while the desired communication pattern of the interoperability model is simply one-to-one (control system to device, or device to control system). Furthermore, the smart home devices intended to be communicated with can be considered (unmodifiable) black boxes. This precludes the necessity of concepts such as special addressing schemes or scoping of shared data. Nonetheless, the importance of providing clear software abstractions for developers is clear, and some concepts presented here (object-oriented MVC design and device aggregation) may be relevant in development of the interoperability model.

3.3 Device taxonomy

As a smart home control system must naturally support a large number of devices with different functionalities or technologies, there must be a way of representing these different properties of the devices supported by the system. Providing a taxonomy of functionalities or other properties and allowing these devices to be categorized according to this taxonomy can facilitate a system handling heterogeneous devices. This is an aspect which is highly relevant for the interoperability model to be developed. The following section will describe a few approaches for such categorization.

ROCob is a network layer for interfacing between a smart home control system and the devices it controls (Papadopoulos et al., 2009). The Java-based system models the devices as bundles conforming to the ROCob device interface using the modular OSGi framework. A device description provides an ID and a type as per the UPnP standard (Presser et al., 2008), as well as a set of functionalities. The following interfaces representing functionalities are provided, adapted from Papadopoulos et al. (2009):

- *Composite*: Used for physical devices with multiple endpoints.
- *Binary Sensor*: Used for sensor-like devices, such as motion sensors.
- *Absolute Level*: Used for physical devices with level properties, such as dimmers.
- *Timed Absolute Level*: Used for physical devices with level properties with an extra timing functionality.
- *Relative Level Control*: Used for devices with direction semantics and read-only values.
- *Relative Level*: Used for physical devices with direction semantics, such as shades.
- *Timed Relative Level*: Used for physical devices with direction semantics with an extra timing functionality.
- *Switch*: Used for on/off functionality of physical devices such as lights and simple home appliances.
- *Battery*: Used for representing a physical battery.
- *Alarm*: Used for devices supporting alarms.
- *Analog meter*: Used for devices which measure analog inputs such as pressure & temperature.
- *Thermostat*: Used for representing a physical thermostat device.

Connect and Control Things (CCT) leverages a JSON/XML based language called Sensor Markup Language (SenML) proposed by Shelby et al. (2016), extending it with logic for controlling actuators (Datta et al., 2014a). In a device gateway as part of the discovery phase proposed together with CCT, actuators identify themselves by sending an SenML message containing their type, allowed range of values (either continuous or discreet), the unit of the values, and a semantic description of the operation supported by the actuator (Datta et al., 2014b). This essentially serves as the taxonomy and allows the user interface to represent the actuators in a meaningful way, but as no enumeration of specific actuator types is presented it is assumed that recognizing and representing devices is deferred entirely to the application using CCT. An alternative actuator extension of the SenML language proposed by the IPSO Alliance is also referenced, where actuator functionalities such as dimmers and buttons are represented (Shelby and Chauvenet, 2012).

Eclipse SmartHome separates the concept of a physical device (a "thing") and abstract functionalities that can be provided ("items"). Devices are supported by defining a thing type in XML, containing metadata such as name, description and possible user configurable properties. All functionality which the device provides is represented as a set of channels, which connect the definition of the thing to its "items". There is a set of predefined item types as described by Eclipse Foundation (2016a):

- *Color*: Stores color information (RGB).
- *Contact*: Stores status of e.g. door/window contacts.
- *DateTime*: Stores date and time.
- *Dimmer*: Stores a percentage value for dimmers.
- *Group*: Nests other items, collecting them in groups.
- *Number*: Stores values in a number format.
- *Player*: Allow control of devices such as audio players.
- *Rollershutter*: Allows control of devices like window blinds.
- *String*: Stores text.
- *Switch*: Allows control of a switch (on/off).

In defining a channel the developer can specify the allowed values of the item (including steps between values in the range, in case of Number or Dimmer types) and whether the item is readable or writable as well. Furthermore, a set of more specific types are available for each channel, among them Alarm, Fan, and Temperature. These channel types define a set of item types for which it is applicable (e.g. Alarm is a Switch item) and an accessible mode (e.g. Fan is readable and writable). The channel types further provide information to the presentation layer of the implementing application on how the device should be represented.

In conclusion, the interoperability model must provide a device taxonomy which is applied to all devices which the system is intending to support. This will allow for device drivers to be written in a more generic way, but will furthermore inform the presentation layer of the widgets it should provide for the user to manipulate the device. The Eclipse SmartHome infrastructure is found to provide a thorough but somewhat scattered enumeration of device functionality types, where it should be considered that most functionality provided to the user by a device can be reduced to a smaller selection. For example, most items storing binary values (e.g. the Contact item) can be seen as variants of the Switch item, while items like Dimmer and Rollershutter provide similar functionality. The precision of the taxonomy has to be thoroughly considered for the implementation of the interoperability model.

3.4 Data model

Providing a common model for data messages exchanged between devices is essential for interoperability. A common data model allows the smart home control system to understand heterogeneous systems which may use different message formats or communication protocols. Achieving this involves specifying a generic data model and translating messages from devices to that format, where the generic data model must naturally be flexible

enough to support all types of data messages expected now and in the future. Multiple approaches have been proposed, some of which will be discussed here.

ROCob uses a common ROCob API that all messages are conformed to (Papadopoulos et al., 2009). For each functionality a physical device supports, a ROCob Device class implements a ROCob Function interface (available interfaces enumerated in the previous section). The Function interface allows other modules in the application to call its methods, thereby allowing communication with the physical device through a commonly understood API.

SenML and the extension for actuators proposed by Datta et al. (2014b) and Shelby and Chauvenet (2012) utilize either a JSON or XML based message format. Applied in a gateway, incoming messages are translated to the common format using device-specific drivers. As seen in Listing 3.2, a light is set to output 100 lux by specifying the value as the `v` attribute and the unit as `u` using the language described in Datta et al. (2014b). The `n` and `t` attributes correspond to the name of the actuator to operate, and the time to carry out the operation offset from the time of receiving, respectively. The message is sent via a HTTP POST to the device-specific *out-proxy* address of the device assigned at the gateway, while the IPSO standard functions similarly but allows PUT, POST, and PATCH messages being sent to the device using a logical path describing the actuator operation (e.g. `/gpio/dimin/1` corresponds to dimmer 1).

Listing 3.2: An HTTP message sent using the SenML extension described in Datta et al. (2014b)

```
POST /mylamp_outproxy  HTTP/1.1
Body: {"v": "100",
       "u": "lux",
       "t": 0,
       "n": "Lamp1"}
```

The **Eclipse SmartHome** framework handles communication by translating messages from the device to the command types supported by the event bus of the framework, and vice versa. This translation is performed by a *thing handler*, which is an OSGi module provided by the programmer. The method `handleCommand` of the thing handler processes commands from the framework whenever one appears on the event bus, and receives as parameters the channel being used to send the command and the command itself (whose type may be, among others described in Eclipse Foundation (2016b), DateTimeType, PercentType, or PlayPauseType). The thing handler may respond to the framework through a callback function. Communication with the device itself is not covered by Eclipse SmartHome, which the thing handler must provide itself by setting up a connection through the protocol required by the device.

Using **binary formats** as opposed to the text based formats presented thus far has been proposed in Zhu et al. (2010), where it is applied in a gateway translating messages between sensor/actuator nodes and a management platform. The message format allows the sender

to use a set of commands (such as toggling a switch or setting light level, temperature, and humidity) modeled as bit fields. These commands are then packaged with information such as source and destination address, payload length, and a Cyclic Redundancy Check (CRC) code for verifying payload integrity. The data transmitted is more compact and quicker to parse at the receiver, which allows the gateway to operate without the overhead of protocols such as TCP or Bluetooth and thus support less powerful devices.

In conclusion, a common data model will allow the interoperability model to represent messages of different types, and thus support the mobile smart home control application to understand and ”speak” different protocols and dialects thereof. Text-based formats should be sufficient, as modern mobile devices have adequate processing power to parse and construct such messages with little performance penalty. It should however be noted that as the target application requires no external gateway for translating messages, the job of the data model is not necessarily one of translating to an explicit message format, but rather about facilitating communication between the application module and the rest of the application in a standardized fashion.

Chapter 4

Interoperability model

This chapter will describe a model for interoperability between heterogeneous smart home devices and a smart home control system. The considerations made during development will be discussed and the finalized interoperability model will then be presented. Lastly, the model is evaluated through a perspective of applicability and usability.

4.1 Development

As part of the development process, a list of concepts required of the interoperability model was developed. The basis for this was the three aspects identified as vital in the State of the Art chapter: the programming abstractions provided to developers wishing to add support for a device, the taxonomy which allows categorization of devices, and the data model used to standardize communication and message exchange between devices and the control system. Subsequently, this section will discuss the choices made in these categories, as well as discussing important relevant concepts such as device discovery.

An important aspect for developing the interoperability model was defining and constraining the purpose of the model in the context of its usage in the target application. Essentially, the interoperability model would serve as a layer between the application and separately written plugins communicating with external physical devices, providing an intermediate interface along with various utilities to facilitate development of both parts. The application should not be required to possess any specific knowledge of the device it is communicating with but rather be able to represent it and communicate with it in a generic fashion. This includes being able to provide the user of the application with widgets in the graphical user interface (GUI) capable of controlling any device in a reasonable way. In the other direction, the plugins communicating with devices should be able to work in a self-sufficient environment, only leveraging the utilities provided by the interoperability model in its communication with and representation of physical devices.

4.1.1 Programming abstractions

While providing programming abstractions is not as relevant as with more complex interoperation like that of wireless sensor network, it is nonetheless still interesting. In particular, abstractions should be provided to satisfy the goal of simplifying the addition of device support for developers.

To develop the relevant programming abstractions, the concept and intended workflow of adding support for a device to a system employing the interoperability model must be considered. Adding such support should involve writing program logic which performs communication between the smart home control system (in this case, a mobile application) and the physical device, without either party knowing how the other works.

While the code would allow two-way communication, the device and the control system should have different responsibilities. The smart home control system must be able to control a device by explicitly telling the device to change some property (such as sending a signal to a light to turn it off or dim to a certain level), while the device should not be able to explicitly instruct the control system to perform a specific action. As an example, it would be unsound to allow a smart light to instruct the control system to unlock a smart lock installed on a home's front door. Instead, it should be left up to the control system to react appropriately to any updates received. With this limitation, the possibility of devices being able to directly communicate with other devices is excluded.

Allowing support for a device to be dynamically or even automatically loaded is simplified if the plugins are not closely coupled with the underlying smart home control system. Since such a plugin may require logic processing messages sent to and received from a smart home device (consider an example of having to parse a bitfield or filter duplicate messages received from a device), it is not sufficient to model the plugin in a declarative language but rather as device drivers written as separate, independent code modules. The developer of device drivers must structure the drivers in a way that allows the control system to interact with them. Allowing the driver to simply conform to an interface thus provides the programming abstraction of the interoperability model. The abstraction provided for driver developers consists of writing an implementation of a set of functions representing common actions such as connecting and receiving messages, without having to consider the logic of the interoperability model (and in extension, the application) which is used to control the smart home device.

4.1.2 Device taxonomy

A device taxonomy is vital to allow devices to be described and represented both visually and logically in device drivers as well as in the smart home control system using the drivers. As such, a device taxonomy must be provided by the interoperability model.

For the purpose of providing a device taxonomy, it is first necessary to define what a device is and what properties realistically could describe it. For this interoperability model,

a device would be defined as a physical device used in the smart home. Specifically, a device would be an entity which is logically a single device; even if the entity is technically separate devices which require separate communication channels yet contained in the same physical housing, it would be considered a device if it was sold as a single device and is intuitively presented to a user as a single device.

In the lifetime of a device seen by a smart home control system, there are two different contexts in which a physical device might be presented to a user. The first context is when a device has been discovered by a driver, at which point the driver knows only a limited set of information about the specific physical device. The second context is when a driver has established a connection with the device, meaning the driver has learned all available information about the device including its current state, and can be controlled by the user. In the former *discovered context*, the driver knows that the device is a product which the driver supports, and can therefore provide pre-defined static data which reflects the types of devices the driver supports. This would allow the GUI of the smart home control system to present limited information allowing the user or system to identify and connect to a device. The static information which could be provided would include at least the vendor and the device model name. In the second, *connected context*, the device is to be considered a *connected device* which could be presented graphically with all available information provided by the specific physical device. Only a connected device would thus be categorized according to the device taxonomy, as some drivers may only be capable of discovering the functionality of a device until after it has connected to it. For a smart light, the light and color effects supported by the specific model may for example be only known after a connection has been established.

To allow the system to establish a connection and thus categorize a device according to the device taxonomy, it must first be discovered by a device driver. Each driver must thus provide a *discoverer*, a process which enumerates the supported devices which can be detected by the driver. The description of a *discovered device* will differ depending on the discovery or communication protocol the physical devices use.

Since a device driver is aware of the particular protocol used, it can enumerate the discovered devices as a protocol-specific description which can be partially understood by the smart home control system and fully understood by the device driver. A device using Bluetooth Low Energy could thus be described with a Bluetooth GUID, a MAC address, and a set of services while a HTTP device could be represented by a URL and a port. These protocol-specific fields of the device description only needs to necessarily be understood by the driver, and should not be utilized by the control system explicitly as such a system should remain entirely device agnostic. Some drivers may produce device descriptions which the system cannot understand, such as for devices which require communication through an intermediary bridge and thus share an IP address, instead using a unique ID to differentiate the separate devices. Another driver may encounter the same situation but may prefer to store both the bridge address and the ID of the specific device encoded as the address field (such as `192.168.0.2/api/light/2`).

A connected device would logically describe a physical device by metadata and functionality. The metadata of a device would consist of generic fields such as name, unique ID, and address. Furthermore, allowing other forms of device specific metadata stored with a connected device can be useful as it allows the driver to save any data it might have to use during communication with the device or in its logic. Such data could potentially be exposed as a settings page of a device where the user can set their preferred settings. A connected device is further described by the functionality it provides, such as if a device supports sending string values or dimming a light. The programming abstractions representing and communicating functionality with the physical device is considered to be the data model, and will be expounded on in the next section.

Several approaches to the programming pattern are applicable when representing devices. In the context of the interoperability model developed here, an object oriented approach seems most applicable since it maps the reality naturally into the model. The choice can thus be made to represents a connected device as an instance of a class with the device metadata as instance variables and a set of methods providing device communication and device discovery. A connected device is thus initialized and categorized by the driver when passing a device description coming from the device discovery process.

4.1.3 Data model

A device the interoperability model wishes to communicate with must be described as a set of functions supported by the device. The functions in turn describe what data can be sent to and received from the physical device. For this interoperability model, the communication does not have an explicit physical data format as in solutions discussed in the State of the Art chapter. Instead, communication is performed bidirectionally between the device and driver, as well as between the driver and smart home control system. Instead of imposing a data format such as structured bit fields or XML, the object system natural in programming is leveraged to provide the data model. Here, classes representing some common functionalities should thus be used or extended by the device drivers.

As stated by the aims of this project, the abstraction of functionality needs to be general enough to encompass a large set of heterogeneous smart home devices. It is thus necessary to consider the specificity of the chosen level of abstraction. If the provided abstractions are very specific, then a large number of device categories needs to be provided. This is the preferred solution in Bluetooth Low Energy which provides 180 different *characteristics* (Bluetooth SIG, 2016), among them temperature control and heart beat monitor, and to a lesser degree in Eclipse Smart Home which provides 32 pre-defined *channel categories* (Eclipse Foundation, 2016c). If instead the provided abstractions are very general, the drivers applying them may have to contain a lot of boilerplate logic extending the abstractions to be able to represent the functionality provided by the device.

A consideration in developing abstractions is if the abstract functionalities should reflect data or function. In the example of modeling a light device which can change brightness

of a lamp, the provided functionality can be abstracted as the data the device expects (e.g. integer values in a pre-defined range) or it can be modeled as the function provided by the device (e.g. dimming of a light). These are two approaches which can give the same outcome, but the difference is by no means trivial. Avoiding mixing the concepts leads to a clearer and more usable abstraction. Eclipse Smart Home mixes the two approaches in the abstractions provided for item types: some abstractions are based on data (e.g. Color, DateTime, String, Number) while some are based on function (e.g. Dimmer, Player).

The functionality abstraction chosen furthermore affects the presentation and control of devices. Providing functionality which primarily reflects data types defers the presentation of the devices largely to the smart home control system, which may be harmful; consider the case where a driver developer wants to allow a user to dim a light. Representing this purely as the data type "number range" gives little indication of purpose to the control system, and its GUI may assume that the appropriate widget is a drop-down menu or a spinner control, which would be unsuitable for the given situation. Data based abstractions (such as ones seen in Eclipse Smart Home) are thus mostly suitable when the interoperability model is applied in applications which know the general type of the devices it intends to allow the user to control. As the interoperability model described here intends to allow the application to be device-agnostic, it is concluded that function-based abstractions are most suitable.

A stated goal of the project was to allow the application implementing the interoperability model to load devices which to the application is unknown, while presenting the devices to the user in an intuitive way. Per previous conclusions, function-based abstractions were found to be preferred for this. Concerning the generality of the abstractions, it was found that creating a few basic general abstractions was sufficient as these could be extended with more specific abstractions at a later stage if the need was found. The chosen abstractions are called *controls*, further indicating the intent of relating the functionality of the device with the widget in the GUI with which it is represented.

As a control reflects a functionality and devices may provide several functionalities often of the same type, each control should contain metadata related to its particular use. In the general case, this includes a label and a description which may be presented to the user in the GUI, as well as a current value which reflects the state of the physical device. Each control would also contain different sets of metadata depending on its type. A dimmer would have a maximum and minimum value, and a toggle switch would have a label for each of its available states. A control would furthermore contain logic for parsing messages received from the physical device which pertain to that specific control, and be able to send updates to the physical device when the control is manipulated by the smart home control system. If faulty input is provided by the system (such as when two controls of a device have conflicting values), the control could reject the update by providing an error message to the smart home control system.

4.2 Results

The resulting interoperability model considers the *device driver* to be the top abstraction, which provides static information about the devices it supports. The driver can discover devices and enumerate them in the form of *device descriptions*. The driver creates a *connected device* when it is told by the smart home control system to connect to a device using its device description. A connected device contains metadata about the specific physical device as well as a list of functionality, *controls*, which the device supports. A connected device further contains logic for receiving messages from a physical device. A visualization can be seen in Figure 4.1.

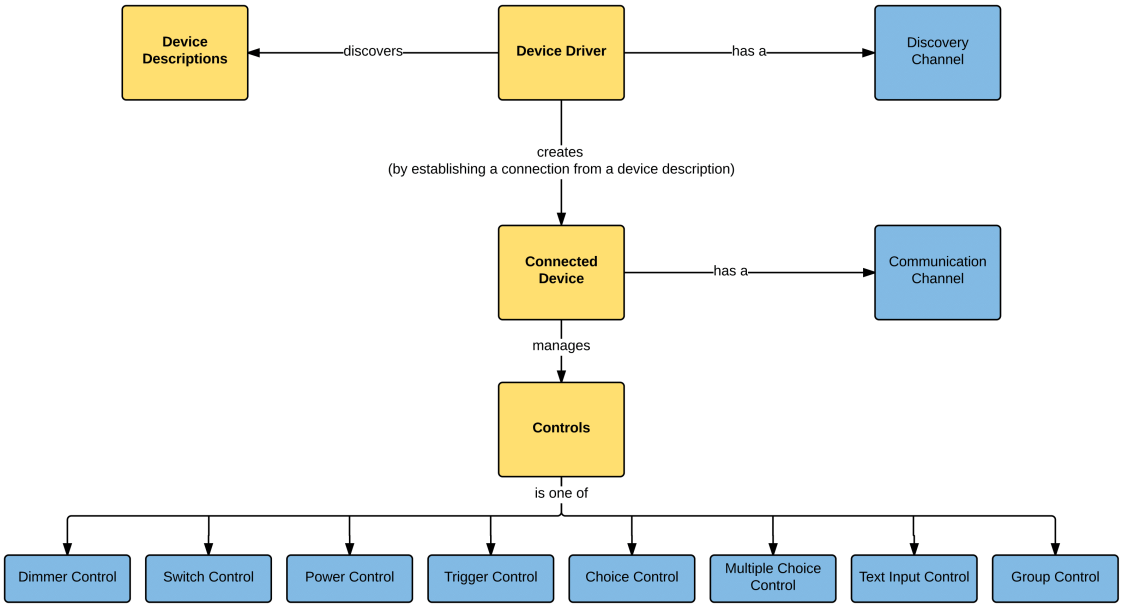

Figure 4.1: A visualization showing the concepts provided by the interoperability model presented in this thesis and how they relate to each other. The entities marked blue are provided by an implemention of the model, while the yellow entities are provided by the developer of a device driver.

As an example case, a device driver for Philips Hue lights is loaded in an application. When the application prompts it, the device driver enumerates all discovered Philips Hue lights by enumerating them as *device descriptions* using the UPnP protocol implementation provided by the UPnP *discovery channel*, each containing the name of the light and its ID. As the device descriptions are received by the application, they are shown to the user. When a user clicks a light, the application calls on the device driver to connect to the light, which the driver does by creating a *connected device* object and instructing it to establish a connection to the light using the provided HTTP *communication channel*. The connected device will then fetch a list of *controls* supported by the device and their current values. When finished, the device driver gives back this connected device object to the application.

The application uses the list of controls to represent a page of appropriate GUI widgets, such as sliders, to the user. As a user manipulates a control, such as a Dimmer Control for the brightness, the application sends these updates to the control provided by the device driver which in turn sends them as HTTP messages to the physical device as appropriate, thereby dimming the light. If the user presses the physical off switch on the light, the light turns off and sends a HTTP message using the established connection to the device driver. The appropriate control receives this message and updates itself with the *off* value and dispatches the update to the application, which at this point reacts by changing the GUI widgets representing the power switch to the off position, and dims the light dimmer slider down to zero.

4.2.1 Controls

The main contribution of the interoperability model is the list of provided control types. These are intended to be able to model functionality of devices, allowing a naive context-inaware smart home control system to represent all functionality of a device in an intuitive way. The specific way of representing a certain control type in the GUI is outside the scope of the interoperability model, but the types are intended to be suggestive of the appropriate widget.

A control contains metadata describing the functionality of the physical device which it models, as well as containing logic for sending updates to both the physical device and the smart home control system, notifying both parties of changes. All controls have a label describing the context of the functionality modeled, and an optional longer description. Controls can allow only the device to update it, or the user as well. Control types may additionally have metadata specific to that type. The control types identified are as follows:

- **Dimmer Control**: A dimmer control is used to model a functionality where the device can have a range of numbers being set which is most accurately shown as a continuous range (even though the device internally may not be able to use continuous values). Examples of this would be the brightness of a lamp or the temperature set on a thermostat. This device control would preferably be presented as a slider widget in the GUI. A dimmer has a maximum and minimum allowed value as well as an optional label for the unit the values are in (e.g. degrees Celsius or Watts). For discrete step-like functionality, the driver implementer can perform a transformation of the user-provided value to the closest supported value.
- **Switch Control**: A switch control models device functionality of a switch which has two states. Examples would be an air conditioning unit which has a heating and a cooling mode. The switch control could be shown to the user as a toggle switch widget. This device control contains a label and value of each state, where the former is shown to the user in the implemented widget and the latter is the value which it corresponds (e.g. a value which is sent to the physical device by the driver).

- **Power Control**: A power control models device functionality of an on/off power switch, either for device power or a specific functionality such as switching on/off a color effect of a smart light. This could essentially be seen as a special case of the switch control, but is separated as it is such an ubiquitous functionality. It is typically presented using a platform-native on/off switch, where the label of the control indicates what it is turning on/off.
- **Trigger Control**: A trigger control represents a device functionality which involves triggering some sort of action, e.g. playing or pausing a film on a smart TV, or opening a window. This control would likely be presented as a button in a GUI with the label as the button text. The trigger can optionally have an icon, which can be used in place of the native button widget in the GUI.
- **Choice Control**: A choice controls models a device functionality where the user is allowed to make a choice among a discrete number of available choices. This is applicable for e.g. choosing the video input on a smart TV or the power setting of a connected microwave oven. This may be presented to the user in the GUI as a list of radio buttons or a drop down list. A choice control contains a list of labels and values of each available choice.
- **Multiple Choice Control**: A multiple choice controls models a device functionality where the user is allowed to make multiple choices among a discrete number of available choices, e.g. a smart home audio system allowing you to choose which speakers your music should play through. This would preferably be presented as a list of check boxes in the GUI. Like a choice control, the multiple choice control contains the labels and values of each available choice.
- **Text Input Control**: A text input control models device functionality where the device allows the user to provide text, e.g. for display on a programmable sign. This control would likely be presented as a text field and a button which sends the update to the driver. A maximum length in characters can be set as a property.
- **Group Control**: A control group is a grouping of several controls and can be used to logically connect the functionality of several controls. This could for example be used when a device driver wants to model a remote control for a smart TV, where trigger controls for changing the channel may be grouped separately from dimmer controls changing the audio volume. The GUI should position the controls of the group together, preferably in a demarcated widget. The only special property of the group control is a list of the controls contained in the group.

4.2.2 Evaluation

The research question pertaining to the interoperability model stated: “Which classifications and abstractions are important when facilitating inter-protocol communication of smart home devices?”.

In this chapter, through a thorough research of the state-of-the-art and by reflecting on the devices which should be able to be supported, it has been found that devices can be classified by the static and dynamic (device-specific) metadata presented here, along with the list of the controls listed above. The model further has the aim of aiding developers in adding device support to the smart home control system the model is applied in. Essentially, this means that the abstractions and tools provided by the model should allow the developer to easily develop drivers, mapping the abstractions provided by the device to the abstractions provided by the interoperability model. To answer the research question and verify that the aims have been reached, the applicability of the model must be evaluated. This evaluation is done by example.

To evaluate the model, it will be applied to a collection of heterogeneous types of devices. Five encompassing categories of devices which are common in the smart home have been identified through researching products supported by systems such as Eclipse Smart Home (Eclipse Foundation, 2016c) and Home Assistant (Home Assistant, 2016). Devices in each such category is used for an evaluation of how their functionality could be modeled by the controls of the interoperability model:

- **Device type**: Smart light
  Description of functionality: The smart light exposes an API allowing a user to set the level of brightness and temperature of the lamp as well as the light color from a set of pre-defined colors.
  Implemented controls:
  - Dimmer Control – label: “Brightness”; minimum value: 0; maximum value: 255
  - Dimmer Control – label: “Color temperature”; minimum value: 0; maximum value: 255
  - Choice Control – label: “Light color”; choices: (“Blue”, 0000FF), (“Green”, 008000), (“Red”, FF0000)

- **Device type**: Smart television
  Description of functionality: The smart TV exposes an API allowing a user to choose video input, audio output, list available channels, set the channel, enable or disable output of 3D video, turn the TV on or off, setting the audio volume, switch to a built-in application, as well as enabling a navigation screen and navigating the options of this screen using direction commands.
  Implemented controls:
  - Multiple Choice Control – label: “Audio output”; choices: (“Digital”, “dig”), (“Analog”, “com”), (“Optical”, “opt”)
  - Choice Control – label: “Video input”; choices: (“HDMI 1”, “av1”), (“HDMI 2”, “av2”), (“TV”, “av0”), (“DisplayPort”, “av3”)
  - Choice Control – label: “Channel”; choices: (“SVT 1”, 1), (“SVT 2”, 2), (“BBC One”, 3), ...
  - Power Control – label: “3D Video”
  - Power Control – label: “Power”

  - Dimmer Control – label: “Volume”; minimum value: 0; maximum value: 80
  - Group Control – label: “Navigation”, controls:
    * Power Control – label: “Navigation Overlay”
    * Trigger Control – icon: “upArrow.png”
    * Trigger Control – icon: “leftArrow.png”
    * Trigger Control – icon: “rightArrow.png”
    * Trigger Control – icon: “downArrow.png”

  Comments: The choice control for channel and application would be initialized only after the driver has fetched the list of available ones from the TV.

- **Device type**: Air conditioning unit
  Description of functionality: The air conditioner exposes an API allowing the user to turn the unit on and off, set the fan speed and temperature, and choose between four operation modes (Fan, Energy saving, Cool, Heat). Additionally, the API provides information about how much power the unit is currently consuming.
  Implemented controls:
  - Power Control – label: “Power”
  - Dimmer Control – label: “Fan speed”; minimum value: 0, maximum value: 10000
  - Dimmer Control – label: “Temperature”; unit: “°C”, minimum value: -10, maximum value: 35
  - Choice Control – label: “Mode”; choices: (“Fan”, 1), (“Energy saving”, 2), (“Cool”, 3), (“Heat”, 4)
  - Dimmer Control – label: “Current power consumption”, unit: “W”, minimum value: 0, maximum value: 1000, controllable by user: false

  Comments: The Current power consumption could also be modeled as a text input control. This would allow the driver implementer more choice in presentation format, but would make it difficult for the smart home control system to e.g. provide automatic actions depending on the value of the current power consumption.

- **Device type**: Window blinds
  Description of functionality: The window blind control unit exposes an API allowing a user to control the blinds in three ways: by pressing the buttons up, down, and stop; by setting the desired level of the blinds; or by pressing the open or close button which fully opens or closes the blinds.
  Implemented controls:
  - Group Control – controls:
    * Trigger Control – label: “Open”
    * Trigger Control – label: “Close”
  - Group Control – label: “Manual control”; controls:
    * Trigger Control – icon: “upArrow.png”
    * Trigger Control – label: “Stop”
    * Trigger Control – icon: “downarrowArrow.png”

  - Dimmer Control – label: “Blinds level”; unit: “%”; minimum value: 0; maximum value: 100

  Comments: The Blind level dimmer control would also be updated by the device driver, e.g. when the user clicks Open it would be appropriate that the level control is set to reflect this as well.

- **Device type**: Carbon dioxide sensor
  Description of functionality: The sensor is able to detect carbon dioxide up to 10,000 ppm in an indoors environment and exposes a simple Bluetooth Low Energy interface where the current reading can be fetched.
  Implemented controls:
  - Dimmer Control – label: “Carbon dioxide reading”; minimum value: 0, maximum value: 10000; unit: “ppm”; controllable by user: false

  Comments: Easy to model, but it is up to the smart home control system to represent the non-controllable dimmer control in an intuitive way and allow the user to e.g. set warnings when it goes above a threshold. A text input control could be employed instead, but would make it difficult to allow the system to set warnings.

By applying the interoperability model to different types of smart home devices, it has been shown that the model can be successfully used in a heterogeneous smart home environment. In particular, it is found that the model can be easily applied to both simple and complex devices with well-defined APIs. There are however some types of devices for which the interoperability model is not designed, namely devices which communicate large quantities of binary data such as cameras or media streaming systems.

Controls for complex data types such as image or video are not part of the scope of the interoperability model since they would require a more complex and specialized model for data transfer. Such controls could potentially be added to support e.g. surveillance cameras, but would require determining the appropriate abstraction level of the provided controls. Some binary data which could be sent would additionally require the smart home control system to support it (e.g. proprietary video codecs) and would thus constrain the devices supported. Such controls would either not provide true interoperability, or require the device driver to transform the binary data to a format understood by the smart home control system and therefore introduce heavy performance critical processing.

Chapter 5

Application module

This chapter will describe the implementation of a software module in an existing mobile application which applies the interoperability model described in the previous chapter. The development phase and the considerations made therein will be discussed. Following that, the structure of the chosen implementation will be presented, and subsequently evaluated with respect to performance.

5.1 Development

From the specified interoperability model, the parts which could be directly implemented had to be identified. As the application module should be implemented in an existing C#-based application which supported Bluetooth Low Energy communication with proprietary light products, reusing existing software infrastructure whenever possible was identified as a goal. For the interoperability model concepts of devices and controls, they were found to be naturally represented as classes.

5.1.1 Dynamic driver loading

A stated goal of the implementation was that device support could be loaded dynamically without the need of going through the standard application update procedure of iOS and Android. Allowing such support required dynamic loading of device drivers. Repurposing the classes representing devices such as lights (including lights organized in mesh networks) which were already available in the application for use in the device drivers was not possible due to limitations of the platforms. These classes were written in C#, and iOS does allow applications which dynamically loads compiled code in its application marketplace, including using Just-In-Time (JIT) compilation (Apple, 2015).

Other approaches had to be considered for dynamically loading drivers, where two dynamic languages were identified as candidates: Lua or JavaScript. Lua interpreters such as

NLua and MoonSharp written in C# allow Lua scripts to be executed without compilation, as opposed to LuaJIT and similar JIT compilers. NLua further provides a unified interface supporting both iOS and Android, which allows Lua objects to be exported to C# and inversely. Running JavaScript is supported on Android natively through WebView which is an application-integrated web browser able to execute invisibly in the background, or via an interpreter running in the Java virtual machine. UIWebView provides similar browser-based functionality in iOS, or JavaScriptCore can be used since iOS 7 to run JavaScript outside of a web view. However, support for the sharing of JavaScript and C# objects through JavaScriptCore seems largely unsupported in Xamarin at the time of writing. Dynamically loading Lua through NLua provides a suitable choice as it has been tested for both iOS and Android and provides the same interface for both platforms, simplifying development. Furthermore, Lua is commonly used for embedded applications and dynamic code loading, which provided a powerful argument.

As a driver needs to be dynamically loaded, the logic which can differ between devices had to be identified so that this can be implemented in Lua. Ideally, this logic should be as constrained as possible, since logic written in C# can be expected to outperform the dynamic code as well as have wider access to code libraries and system services such as communication stacks. Since the class representing a device will contain metadata of a physical device as well as logic for handling received messages, it needs to be implemented in the Lua drivers. The classes which represent different types of controls must also be implemented in Lua, as they will contain logic for updating device data either locally by communicating with the application graphical user interface, or by communicating updates of its state to the device. A service for discovering devices of the type the driver supports will also have to be written in Lua for the same reason.

Since the drivers would be written in Lua while the rest of the application in C#, providing a layer between NLua and the application would allow for a modular implementation. Specifically, application code should not have to be concerned with the structure of the Lua drivers but should merely interact with C# code which abstracts away those details. Essentially, there needs to be a Lua runtime class which can provide this interface between Lua and C# code. This runtime should know the basic functionalities of drivers, such as listing discovered devices, allowing a device to be connected to or disconnected from, and listing the controls of a specific device. Whenever objects such as controls or devices need to be exposed to the application code, the runtime class will have to translate the Lua objects of the driver to C# equivalent wrapper objects. This would provide an interface not dependent on the Lua implementation and further allows the Lua runtime to be switched to the alternative or complementary JavaScript runtime if that provides a more suitable alternative going forward.

The dynamically loaded device drivers imply that developers are provided with flexibility in implementing and structuring drivers, as they are only constrained by needing to expose a common interface to the C# code. While the drivers are thus free to implement controls specific to their supported devices, they are essentially restricted in how the C# code can

interact with them. A control for more complex devices (such as a smart TV) or specific data (such temperature or humidity) must thus be able to be expressed in the context of the controls which the interoperability model provides (in extension, the controls which the application can interpret).

5.1.2 Communication channels

Providing a common interface for communicating over different protocols and media allows drivers to reuse implementations without having to provide their own, as is required in e.g. Eclipse SmartHome. The pre-existing `IAdapter` interface designed for Bluetooth Low Energy (BLE) communication was repurposed for this use case. The class provides event handlers for device discovery, connections and receiving messages. This code was written in C# as Lua does not provide much in terms of communication stacks while making it more difficult to deploy and evaluate performance and stability.

There are multiple channels which may be of interest for communicating with smart home devices, such as Bluetooth, Bluetooth Low Energy, and IP-based protocols on multiple levels of abstraction (TCP/UDP, HTTP, REST). Providing an abstraction suitable for all these protocols is difficult as the protocols are quite different in how they expose the data from the devices.

At a low level of abstraction, most devices and their exposed data could be abstracted as a key-value store, as they use some form of identifier or address (key) to fetch specific data (value). However, this level of abstraction quickly becomes infeasible; in HTTP the key would include (at least) the URL of the requested resource, the method (e.g. GET, POST) and optionally a payload. In an established Bluetooth Low Energy connection however, the data provided by the device is modeled as *characteristics*, which would serve as the key along with an optional payload. Furthermore, providing a complete abstraction for communication channels would hide some protocol-specific data or process which may be essential to some drivers, such as the binary data provided by Bluetooth scan records, or the difference between a paired and a connected Bluetooth device. Such a low-level abstraction is thus not suitable for the implementation.

Protocol-specific metadata, e.g. characteristic for Bluetooth Low Energy devices, needs to be exposed to the device driver which is using a communication channel. It is thus suitable to leverage the dynamic nature of Lua to allow the communication channels to largely provide their own specific interface appropriate to the underlying communication protocol. The essentials such as device metadata and connection status can be part of a general interface which all implementations of the IAdapter class must fulfill.

Numerous protocols are used in smart home environments and are thus candidates for being implemented as communication channels. Such protocols include but are not limited to:

- Bluetooth

- Bluetooth Low Energy
- HTTP
- REST
- TCP
- UDP
- ZigBee
- Z-Wave
- 6LoWPAN

Some of these protocols may build on other protocols. For example, REST leverages HTTP but provides a simplified interface which does not use persistent connections nor state. For this case, an implementation of a REST communication channel could thus leverage an HTTP communication channel internally.

5.1.3 Discovery channels

For some protocols, discovery is heavily intertwined with communication – an example would be Bluetooth, where any implementation of the Bluetooth stack is expected to provide both communication and device scanning, as there is only one unified way of discovering Bluetooth devices (disregarding driver specific heuristics for identifying and authenticating devices). For other protocols there may not exist an inherent way of discovering devices, such as TCP/IP and HTTP. Discovery for these protocols may be entirely unsupported (a user must know the IP address directly) or can happen by sending broadcast TCP or UDP messages for which the device responds with its identity. A common example of the latter example is the protocol employed for discovery in Universal Plug and Play (UPnP), called Simple Service Discovery Protocol (SSDP).

The discovery protocol a device uses is thus not always related to the communication protocol it uses, and as such it is suitable to provide separate channels for discovering devices. A device could for example use a proprietary HTTPS-based protocol for communication, but allow discovery through SSDP, and therefore require both a communication channel and a discovery channel instance in its device driver implementation.

Among the communication protocols listed in the previous section, Bluetooth (including Low Energy), ZigBee, and Z-Wave provide a unified discovery and communication channel. For the protocols which do not provide such an interface, SSDP, Android Network Service Discovery and zeroconf are separate protocols which may be relevant.

5.1.4 Device Drivers

Device drivers implement the code allowing the application to send messages to and receive messages from a physical device, providing a common interface for all devices to the smart home control application. The drivers represent the intended goal of interoperability in the smart home, and as such the development of the device driver ecosystem constitutes a substantial part of the implementation of the interoperability model.

Device drivers are implemented in Lua which does not support object-oriented programming natively, which is instead commonly implemented either manually through *metatables* or by using a third-party library. The object-oriented paradigm is suitable for both representing the devices and controls themselves as well as the C# helper classes such as the communication and discovery channels. Further, a device driver should logically be allowed to manage several connected devices. Implementing this without an object-oriented paradigm involves representing each device as a Lua primitive type table (the only collection available in the language), at which point it can be equated to an ad-hoc implementation of object-oriented programming. Thus, an established third-party library such as *yaci* should be used for representing devices and controls as objects.

The implementation of a device driver should follow the concepts of the interoperability model, where a device is described as a set of metadata and a set of controls. The device should be implemented as a class, which holds as its fields the metadata and includes a method which fetches the controls of the device. As a driver may handle multiple models of devices (although typically from the same product line), these controls could be initialized only after a connection has been established with the device by the driver. The purpose of the device class also includes parsing messages received by the device, and dispatching the message to the corresponding control.

The controls are classes whose responsibility is to handle the updates which are sent to it. Updates of the state of a control may come from the application, in which case the control should provide logic for writing this value to the physical device. If the state update originates from the physical device, the control should contain logic for updating the value locally and informing the application of this change. Additionally, control classes contain all metadata which the interoperability model describes for their corresponding type.

For evaluation purposes, a few device drivers will be implemented. For ease of testing, the most applicable devices to be supported would be lights from Plejd which use a proprietary protocol running on top of Bluetooth Low Energy, and Philips Hue lights which utilize a REST interface for communication to a bridge, which in turn uses ZigBee to communicate updates to lights. As these are light products, they will integrate well into the existing infrastructure of the application which is currently targeted only to Plejd products.

5.2 Results

This section will present the application module as implemented. The implementation will be discussed and the research questions stated in the introduction will be answered by applying the findings herein.

5.2.1 Implementation

The application module has implemented and tested for Plejd lights and Philip Hue lights. An overview of the components of the implementation can be seen in Figure 5.1. This implementation will be presented in this section before it is evaluated in section 5.2.2.

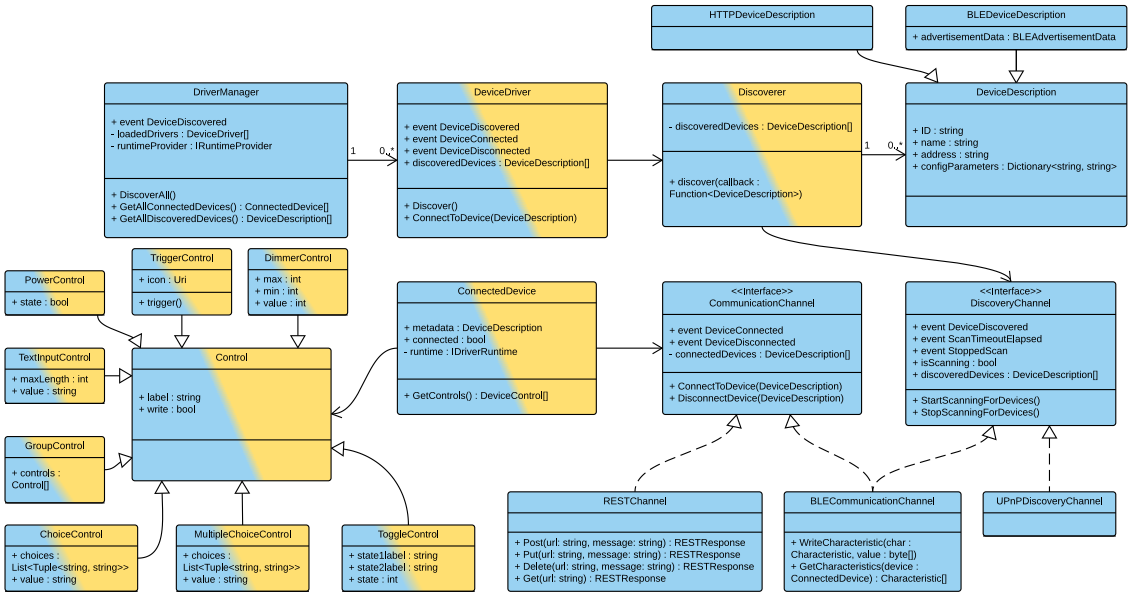

Figure 5.1: A UML class diagram of the implemented application module. Classes implemented in C# are shown in blue, while blue/yellow signifies Lua classes wrapped by C# code.

Essentially, the implementation consists of two related parts: the C# code and the Lua code. All Lua code which is exposed to the application is wrapped by equivalent C# classes to ensure that interaction between the application and this interoperability module is runtime independent. The C# code additionally exposes some platform-native helper code to the Lua drivers, such as fetching discovery and communication channels, printing output for debugging, fetching information about the user logged into the application, and showing modal dialogs to users.

The relationship and communication between Lua and C# code is visualized in Figure 5.2.

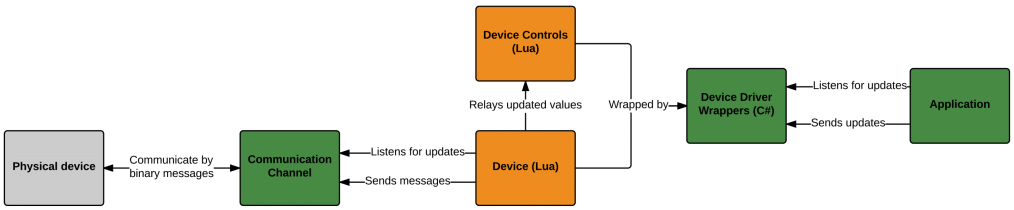

Figure 5.2: A visualization of how communication between physical devices, communication channels, Lua drivers, controls, and the application is performed. Code which is provided by Lua device drivers is marked orange, while C# code is green.

The implementation can load device drivers which have been bundled with the application or from the file system through the `DeviceManager` class. An instance of the `DeviceManager` class exposes a C# wrapper of a Lua driver, and can be used by the application to discover devices. The Lua driver performs discovery using a C# discovery channels (Bluetoth, Bluetooth Low Energy, or UPnP) which is retrieved from the application by calling the exported static C# method `getDiscoveryChannel(type: String)`. The discovery channel returns discovered devices through a callback to the Lua driver, which the driver if needed performs heuristic identification of by e.g. comparing the name of all discovered Bluetooth devices to the one expected by the driver. When the driver discovers a supported device, it returns it to the C# code via a callback. The application can then enumerate the discovered devices as `DeviceDescription` objects and connect to any such physical device by passing in the description to the `DeviceDriver`. When a connection is requested by the application, a separate Lua runtime is started for the device driver to run in without conflicting or interacting with the global environment of the runtimes of other drivers. At this point, the Lua driver may perform some initialization such as populating the list of `Control` objects and setting metadata instance variable. This discovery/connect procedure is shown in greater detail in figure 5.3.

When a connection has been established by a device driver, the application is ready to exchange messages with the physical device. The device driver connects to the physical device through a `CommunicationChannel`, registering itself for callbacks when the channel receives a message from the device. A Lua driver processes these device messages by parsing them and dispatching the received updated state to whichever Lua control object it pertains which in turn informs the application through a callback. If a user manipulates the widget in the GUI representing a certain `Control`, the application updates the state of the C# `Control` object which in turn dispatches a message to the underlying Lua control. The Lua control then sends this information to the physical device, but may at this point instead send an error message for incorrect state or filter messages if they are erroneous or e.g. too frequent. This message passing procedure is shown in the sequence diagram in Figure 5.4.

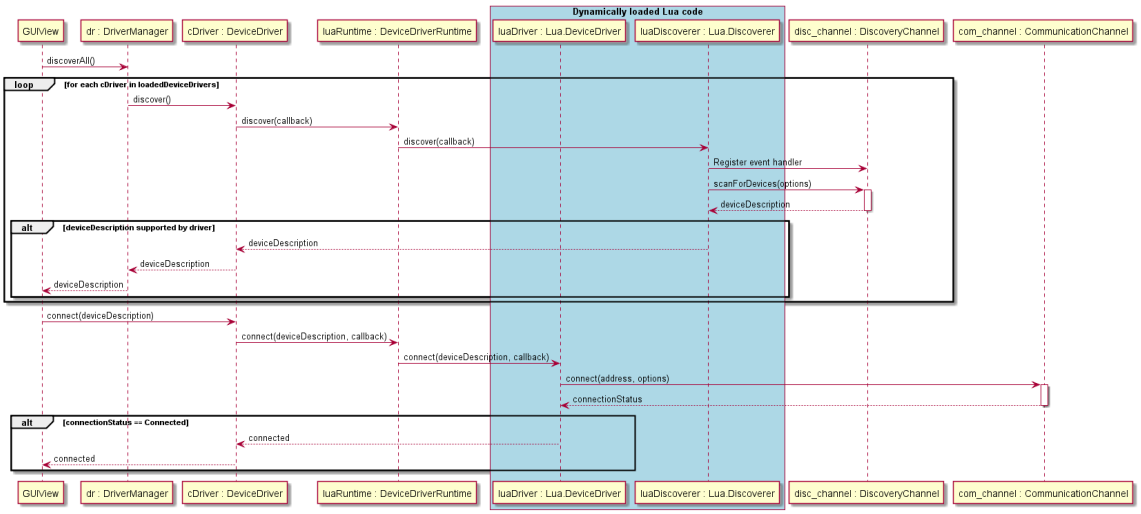

Figure 5.3: A sequence diagram showing how the discovery/connect procedure is performed in the workflow of the application.

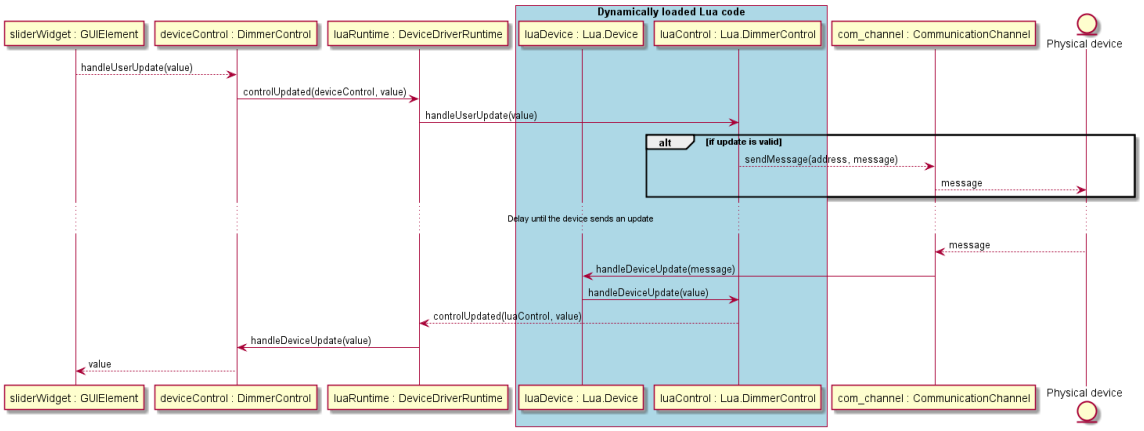

Figure 5.4: A sequence diagram showing how a Lua device driver receives messages in the `handleDeviceUpdate` method and sends messages with the `handleUserUpdate` method.

Device drivers for Plejd dimmer devices and Philips Hue have been implemented. An abbreviated Lua implementation of a Philips Hue device driver is shown in Appendix A.

5.2.2 Evaluation

The implementation has been applied in a real-world scenario by a user controlling Philips Hue and Plejd lights through GUI widgets representing the functionality of the devices. The application has been found to work, but it is further valuable to evaluate whether

the implementation performs well. The chosen evaluation method involves both simulated and real-world testing environments, where simulated environments will be applied for tests whose result is difficult to evaluate visually through inspection of the light devices. This section will describe the tests conducted and evaluate the applicability of the implementation.

Application-to-Device communication

The first test consisted of a test program which sends messages to both Plejd and Hue lights connected through Lua drivers to create a light show. This evaluates message throughput and timing from the application to devices through the Lua drivers, where the success condition is a steady change of light brightness and color without noticeable jitter or message grouping. A program for changing the state of the connected lights was written and added to the application code. Connected to the application were ten Plejd lights organized in two meshes (a configuration where the application can communicate updates to any light in a mesh by talking to one predefined light) and five Hue lights sharing a single bridge. The program would randomly choose one of the connected lights to which it would send an update switching from a high to a low value or vice versa on the dimmer control modeling brightness or color of the light. The periodicity of this process depended on a timer which was set to fire at regular intervals. The effect produced in light shows for different values of timer intervals can be seen in Table 5.1.

The first test shows that Plejd lights seem to work without issues, which can be assumed to be due to the use of Bluetooth Low Energy which is by its specification intended to support messages being exchanged every 0.0075 second (Townsend et al., 2014, Chapter 1), or equivalently 133 messages per second. For the highest load in the test of 1000 messages per second, each Plejd light mesh is expected to receive one third or roughly 333 messages per second, which interestingly does not seem to cause any visually noticeable issue.

The reason why Philips Hue lights seem to have experienced issues in the first test is further researched by excluding the Plejd lights and running the test again. The results show that stuttering starts showing occasionally at 20 packages per second sent to the Hue bridge. Debugging indicates that this delay originates in the Lua driver as it is comfortably capable of sending messages at even rates of 1000 packages per second. The issue seems to manifest itself on the Philips Hue bridge, which the Hue specification indeed mentions does not support more than 25 commands per second otherwise buffering may take place (Philips, 2016).

Device-to-Application communication

The previous test evaluated sending messages from the application to a physical device using the implementation, but it is also relevant to evaluate the performance of messages sent from smart home devices to the application. Such a test will verify that the solution of using Lua device drivers can perform well under heavy load. The Lua device for Philips

Timer fire rate	Updates per second	Summary
0.2 s	5	No issues, smooth but slow dim changes
0.1 s	10	Smooth dim changes
0.05 s	20	Smooth and fast dim changes, no issues
0.01 s	100	Plejd lights work smoothly, Hue seems to have issues with grouping messages
0.001 s	1000	Plejd lights work smoothly, Hue seems to have issues with grouping messages

Table 5.1: A test of sending messages randomly by timer to 10 Plejd lights organized in two light meshes and 5 Hue lights connected to the same bridge.

Hue is reused, and as a base line, simple code for receiving TCP packets following the pattern of the device adapters from the original application code base is developed. This code thus follows the device-native approach for receiving TCP packets, which serves well as the base line to test the performance of the Lua drivers of the interoperability model.

For this test, simulated smart home devices are required to allow control of the volume and contents of the messages sent. A simple REST web service is built, which provides a subset of the functionality expected by the Philips Hue driver. The web service is programmed to send TCP messages containing Philips Hue JSON payloads as quickly as possible to the application once a connection has been established. The number of messages transmitted by the web service is a configurable parameter `numMsg`, which is increased iteratively to subject the application to heavier loads.

The test is performed by deploying the web service on the local network, as this minimizes latency and is the most likely case for smart home devices using the HTTP protocol. The `numMsg` parameter is started at 10 and for every test is increased by an order of magnitude while the duration of the test run is considered feasible. Each test is repeated ten times, and the average duration of each test is noted both as the application is configured to use Lua device drivers as well as platform-native processing. The results are shown in Table 5.2.

From these results, it can be seen that the overhead for a Lua driver to process a message and dispatch it is quite insignificant. The speed of the processing by the native solution grows somewhat as loads are higher, but the difference in processing speed between the native and Lua solutions remains under 5%. It is concluded from this test that while there is some overhead for Lua drivers to process messages and dispatch them to the application backend, the difference in performance remains insignificant under even very high loads

Messages (#)	*Lua driver*		*Native*		Difference (%)
	Duration (ms)	Speed (messages/s)	Duration (ms)	Speed (messages/s)	
10	303	33.0	293	34.1	3.41%
100	2034	49.1	1971	50.7	3.20%
1000	17273	57.9	16733	59.8	3.23%
10000	157678	63.4	152023	65.8	3.72%
100000	1546231	64.7	1485012	67.3	4.12%

Table 5.2: The results of a test measuring the time taken for the application to process a given number of messages sent to it by a simulated smart home device. The difference in performance (total time and messages processed per second) between the dynamic Lua device driver system and a native implementation is shown.

and should thus not affect the user experience in any significant way.

5.2.3 Research questions

Two research questions pertaining to the implementation were stated. This section will answer and reflect upon these questions.

Dynamic loading of device support

The first research question stated, ”How can plugins for protocol support be loaded on the fly, or even automatically?”.

In the context of the cross-platform mobile application, it was found that protocol support would most suitably be implemented as device drivers which could then be loaded on the fly through dynamic code loading. Lua was found to be a good candidate supported by both platforms for this dynamic loading, and further providing a performant and flexible solution for driver implementation. This serves as an answer to the first part of this question.

The answer to the second part of the question, how device support could be loaded automatically without user intervention, has not been part of the implementation and represents a related but separate problem. For drivers which are not bundled with the application but must be downloaded from a separate source, this is not a trivial problem. Extending the current implementation, each driver available through the central device driver repository could provide a secondary, minimal discoverer. This minimal discoverer would only indicate the discovery channel the driver uses and provide a simple boolean condition which can only use the device description returned by the discovery channel to identify if the device is supported. Such a solution would be sufficient for most device drivers which have a simple discovery process and also remain highly performant for a large number of available drivers. Using this solution, a user could use the application which could then find all nearby devices for which there exists a device driver, and then connect to and allow the user to control the device on the fly without explicit user intervention.

Reliable and performant communication

The second research question stated, ”How to allow reliable and performant communication with heterogeneous devices from a mobile device?”.

Multiple aspects have been considered to address this issue, such as implementing communication and discovery channels in C# leveraging platform-specific communication libraries and thus allowing the device communication itself to be as performant as possible. Nonetheless, message parsing and heuristics being written in Lua can be expected to introduce some performance overhead, but this was determined to be largely insignificant in the context of the application and the types of devices that intends to communicate with.

Reliability-wise, the implementation has some drawbacks as some stability issues have been experienced. These issues are not due to Lua itself but the NLua library available for iOS and Android development through Xamarin which has been causing some crashes due to memory segmentation faults. A solution using a more up to date NLua implementation would be preferable, or potentially replacing the Lua runtime with a platform-native JavaScript runtime.

Chapter 6

Discussion

This thesis has presented the development of an interoperability model ultimately intending to increase the viability of the smart home. An implementation of the interoperability model aims to show that the model is applicable. In light of these findings, this chapter intends to discuss the development process and results. Furthermore, future work and ethical considerations will be discussed.

6.1 Interoperability

In developing the interoperability model, the state of the art research was seen as being quite valuable. While much research focused on routing and low-level communication, some ideas were found to shape the thinking around the intended interoperability model. The intention of this thesis was to present a model which could serve as a framework for developing smart home control systems allowing communication with heterogeneous devices, and while research showed that there were some commercial products which claimed to provide interoperability, they did so by using proprietary device drivers developed in-house instead of allowing third-party developed drivers. Some concepts employed in Eclipse Smart Home were however influential in the design of the model.

The interoperability model has been found to be highly applicable, but there are some types of controls which have been found difficult to handle. Controls for complex data types such as video have been previously discussed, but the opinion of the author is that these controls should not be provided by the model as they increase the complexity of the smart home control system using the drivers and breaks the intended interoperability. Instead, such situations should be solved by allowing the interoperability model to facilitate connections between a device such as a surveillance camera and another device such as a screen.

Another difficult situation for the interoperability model is grouping of controls, especially if fine-grained positioning of controls is required. In providing this, there are potential issues in allowing a device driver to explicitly position GUI widgets on a pixel level, as

well as it being difficult to implement in a fail-safe and platform independent way. A suggested solution could be to provide some common groupings such as columns and rows, or to provide purpose-specific controls such as a *direction control* consisting of correctly positioned arrows for sending direction commands to a smart home device.

The choice of constraining the interoperability model to not allow direct intercommunication between devices has certain downsides and may limit the actions of some devices. As a device driver cannot explicitly tell another device driver to perform an action, the smart home control system must explicitly support this intercommunication. If the supported intercommunication is highly device-specific, it would go against the idea of allowing the smart home control system to be inaware of the devices it controls. While direct intercommunication between device drivers can allow for interesting and creative behavior, it also negatively affects the privacy and security of the smart home, which was seen as the priority.

While the interoperability model may require revision as the concepts of the smart home progress and change, it is considered by the author to be a good base. The results and the reflections presented in the Development section of the Interoperability model chapter serves as a good guideline for further development of the model by e.g. adding more controls. To make the interoperability model even more viable, it may serve it well however to be rewritten as a standards document.

The resulting interoperability model is considered by the author to be well devised and applicable, and especially interesting as it allows device drivers to largely influence the presentation of devices without being able to explicitly control it. The controls mark their intent (function) instead of their data, allowing the graphical user interface to choose appropriate widgets to represent them. This concept is the key to allowing the smart home control system to control heterogeneous smart home devices while being largely unaware of the devices it is being used to control. The result is a good balance between drivers being powerful enough to influence the smart home control system to intuitively present devices, and being quite simple and terse to implement.

The contribution of this thesis to the state of the art could be considered to be a well-defined model which can be used in applications which are largely unaware of the types of devices they control. In contrast to existing solutions, the interoperability model presented in this thesis also provides implemented channels for communication and device discovery, allowing device driver developers to focus on the logic used in communicating with its devices. The solution is further suited to being applied in smart home control systems deployable on smart phones.

6.2 Future work

The interoperability model and especially the application module implementing it are works in progress. Some of the features which may be considered for future development are listed in this section.

- The controls of a device are described by text labels and descriptions, and the drivers can produce error messages or require user input. The GUI of the application will show these messages to the user, but there is not yet support for any localization of these messages.

- A device setup process could be represented as part of the device drivers. Some devices require user-guided setup (i.e. naming, providing username and password for login, or creating accounts), which could be represented in an abstracted way. A declarative solution could be such a way, allowing the driver to indicate the data it requires for setup of a device and the type or which ranges of values are allowed. The setup process could be represented as decision trees, allowing processes where a device can either allow the application to authenticate itself or instead allow the user to create an account before proceeding.

- Some drivers may require persistent storage where data such as memorized authentication information or user-preferred settings can be stored. An aspect to consider is whether this feature should be part of the interoperability model or solely the application implementing it. An argument could be made that it should be on the application side, integrated into the presentation of the device setup suggested above where the known information could be remembered and automatically entered into the GUI. Alternatively, a simple key-value store could be exposed to device drivers, allowing for other types of simple persistent data storage.

- While the controls of the interoperability model are moderately coupled with the GUI widgets used to present them, there are cases where there are multiple suitable choices of widgets. A choice control could for example be represented as a dropdown menu or radio buttons. For most cases it is suitable to defer to application logic, which could e.g. use radio buttons if there are four choices or less. However, in some cases it would be preferable for the drivers to be able to request a widget used to display it, which the application may approve or deny. This is preferable to simply exposing all widgets available to the device driver, since some widgets may be platform-specific (while the drivers should always be platform-independant) or simply not fit well into the viewport of the GUI. Thus, each control of the interoperability model could be extended with an instance variable which allows them to request a specific widget.

6.3 Ethical considerations

The developed interoperability model concerns smart homes, and thus raises some ethical questions related to automation and smart homes in general. These are relevant to discuss since the model could increases the viability and, consequently, the adoption of smart home devices. This section aims to briefly bring up some of these issues.

Positives

- The smart home raises quality of life for persons with mobility issues and the elderly.
- Automation of lighting, ventilation and environmental controls in homes can minimize their environmental impact.
- It can be argued that since the control system using the interoperability model is merely relaying information between devices, its manufacturer could or should not be held responsible for this data or what the devices are used for, even in cases where it can abet malicious behavior.
- A protocol agnostic system can help alleviate vendor lock-in that can be experienced with other vendor specific systems.

Negatives

- A smart home control system may be used to increase the environmental impact of a home. In fact, the comfort and ease-of-use can make it more effortless to increase one's negative effect on environment, by for example setting the heating to high. Even well-grounded ideas, such as switching lights on when one is away to discourage burglaries, can contribute to this. In short, ease-of-use and automation of tasks makes it simpler to make bad choices, as well as good.
- There are inherent security issues with a powerful smart home control system, allowing the possibility of someone "hacking" into your home. This can give great power to malicious entities.
- Manufacturers may have a financial stake in their smart home products being used in conjunction with their own control systems, and rely on consumers investing in their entire infrastructure. However, the interoperability model does not intend to reverse-engineer or break protocols maliciously, but rather apply open and available protocols.
- If the system is used for safety-critical automation such as for locks or alarm systems, it can be catastrophic if the system is unreliable. If messages or data is lost, the manufacturer may be liable for damages incurred because of this unreliability.
- An increased adoption of the smart home can cause skill degradation and dependency, especially as more substantial yet menial tasks are automated by systems. While this could be considered a positive, in an admittedly unlikely mass failure of these systems it could prove disastrous. For safety-critical applications such as health monitoring or even doctor's duties the effect of this would be quite apparent. In heavily automated homes, some tasks may even rely on the cooperation of the smart home control system, and may be impossible to do manually.

- A smart home inherently and necessarily stores and processes large amounts of data on the activities and routines of the people living under the system. As such, the system can be an accomplice to surveillance conducted by malicious entities such as thieves or even governments. Such malicious users could use the system to surveil users to great detail, since all their home and personal data (calendars, health records, bank accounts) can potentially be plugged into it. This may even lead to an inadvertent loss of privacy if bugs in the system accidentally give out a user's information to others sharing the system.

Chapter 7

Conclusion

This Master's thesis has described the development of an interoperability model for allowing control of devices of the smart home. The stated goal was to make the interoperability model general enough so that it could allow heterogeneous devices to communicate with a system implementing the model. The resulting model has then been implemented in a mobile application using Xamarin and C#. The aim of the implementation was to provide a seamless user experience by dynamically loading support for smart home devices, as well as providing reliable and performant device communication.

The developed interoperability model allows device support to be added by writing device drivers. A device driver is structured according to a contract which the interoperability model can understand, describing a physical device with static as well as device-specific metadata. The functionality of a smart home device is modeled as a list of controls which map the functionality provided by the device to the functionality understood by the interoperability model. The controls provided by the model are dimmer controls, switch controls, power controls, trigger controls, choice controls, multiple choice controls, text input controls, and group controls.

The implementation of the interoperability model as a module in a mobile application uses dynamic code loading to load device drivers written in Lua. The application module has been structured in a way to essentially not be dependant on the rest of the application, and as such it hides the implementations of the device drivers from the rest of the application. The application logic has been extended to render elements in the graphical user interface to allow user control of various devices. Device drivers for smart lights from Philips and Plejd have been implemented and tested.

The aim of allowing heterogeneous smart home devices to communicate using the interoperability model has been largely reached. Evaluation by applying the abstractions of the model to various types of devices found the model to be highly applicable. However, it is noted that the interoperability model does not support devices which communicate large

quantities of binary data such as cameras or media streaming systems. It is found that using Lua for the aim of providing a seamless dynamic driver loading has been largely successful, but it is noted that instabilities in the chosen code loading library NLua needs to be addressed. Lastly, the aim of providing performant communication has been fulfilled as device drivers were found to perform well in a high traffic test environment.

The interoperability model is deemed by the author to be relevant to the smart home field through its applicability and ease-of-use. This has been shown through the evaluation of the interoperability model by theoretically applying it to heterogeneous device types, along with the successful implementation of device drivers for two smart home devices using different protocols. It is however noted that the model should be further tested in real-world scenarios with a larger number of devices. Furthermore, as the evaluation of this sort of system is inherently practical, there may very well exist, or come to exist, smart home devices for which the provided controls or even the fundamental principles of the model do not function.

Appendix A

Philips Hue device driver

This appendix presents a simplified implementation of a device driver for Philips Hue devices, showing the essential structure used by driver developers. This device driver, written in Lua, can be loaded in the application module described in Chapter 5. It consists of three modules: the *discoverer*, the *controls*, and the main driver class which is the *device*.

A.1 Device

```
require "yaci"
require "HueControls"
require "HueHelpers"
json = require "json"

name = "Hue"
vendor = "Philips"

Driver = newclass('Driver')
function Driver:init(deviceDescription)
    self.deviceDescription = deviceDescription
    self.status = "disconnected"

    self.controls = {on=HueSwitch:new{label="Power", write=true,
       device=deviceDescription},
                     bri=HueDimmer:new{label="Brightness", min=1,
                        max=254, write=true, device=deviceDescription},
                     hue=HueDimmer:new{label="Hue", min=0, max=65535,
                        write=true, device=deviceDescription},
                     sat=HueDimmer:new{label="Saturation", min=0,
                        max=254, write=true, device=deviceDescription},
                     ct=HueDimmer:new{label="Temperature", min=153,
                        max=500, write=true, device=deviceDescription},
```

```
                    effect=HueSwitch:new{label="Color Loop Effect",
                        write=true, device=deviceDescription,
                        onValue="colorloop", offValue="none"}}
end

function Driver:handleDeviceMessage(message)
    parsed = json.decode(message.Content)
    if message.StatusCode ~= 200 or parsed["error"] ~= nil then
        error("Warning, got error device message: "..message.Content)
        return
    end

    for key, control in pairs(self.controls) do
        control:handleDeviceUpdate(parsed["state"][key])
    end
end

function Driver:handleDeviceLost()
    print("device "..self.name.."
        ("..self.deviceDescription.Address..") lost!")
    self.status = "disconnected"
end

function Driver:disconnect()
    print("device "..self.name.."
        ("..self.deviceDescription.Address..") disconnected")
    self.status = "disconnected"
end

function Driver:connect(onConnectCallback)
    local channel = getCommunicationChannel("REST")

    endpoint = self.deviceDescription.Address.."/api/"..
                helpers.getUserName().."/lights/"..self.deviceDescription.ID
    responseTask = channel:GetAsync(endpoint)
    responseTask:ContinueWith(function (task)
        self.handleDeviceMessage(self, task.Result) end)

    -- Since Hue uses REST we don't use connections. We could have a
        periodic check here for light updates, but we don't yet.

    print("device "..name.." ("..self.deviceDescription.Address..")
        connected")
    onConnectCallback(self.deviceDescription)
    self.status = "connected"
end
```

A.2 Discoverer

```
require "yaci"
require "controls"
require "HueHelpers"
json = require "json"

Discoverer = newclass('Discoverer')

deviceDiscoveredCallback = nil

function Discoverer:init()
    self.channel = getDiscoveryChannel("UPnP")

    self.channel.DeviceDiscovered:Add(function (sender, args)
        self.deviceDiscovered(self, sender, args) end)
end

function Discoverer:discover(onDiscoveredCallback)
    deviceDiscoveredCallback = onDiscoveredCallback
    self:discoverUsingUPnP()
    self:discoverUsingNUPnP()
end

function Discoverer:discoverUsingUPnP()
    self.channel:StartScanningForDevices()
end

function Discoverer:discoverUsingNUPnP()
    restChannel = getCommunicationChannel("REST")
    response = restChannel:Get("http://www.meethue.com/api/nupnp")
    if response.StatusCode ~= 200 then
        print("Could not discover Hue bridges using NUPnP")
        return
    end
    devices = json.decode(response.Content)
    for i, device in ipairs(devices) do
        print("Found NUPnP Hue bridge: "..json.encode(device))
        self:bridgeDiscovered('http://'..device.internalipaddress)
    end
end

function Discoverer:deviceDiscovered(sender, args)
    prefix = "philips hue bridge"
    name = args.ModelName:lower()

    if name.sub(0,prefix:len()) == prefix then
        self:bridgeDiscovered(args.Address)
    end
end
```

```
function Discoverer:bridgeDiscovered(bridgeAddress)
    restChannel = getCommunicationChannel("REST")

    response = self:getLightsFromBridge(bridgeAddress, restChannel)
    if response.StatusCode ~= 200 then
        error("Got error "..response.StatusCode.." when fetching
            lights")
        return
    end
    parsedResponse = json.decode(response.Content)

    errorResponse = self:getErrorFromResponse(parsedResponse)
    if errorResponse ~= nil then
        unauthorizedUser = 1
        if error.type == unauthorizedUser then
            -- if the user is not allowed, we assume the user has not
                yet been created yet
            -- (i.e. is not on the Hue "whitelist"), and thus we
                create it and repeat the request
            self:createUser(bridgeAddress, restChannel)
            response = self:getLightsFromBridge(bridgeAddress,
                restChannel)
            parsedResponse = json.decode(response.Content)
        else
            print("Got unexpected Hue error code "..error.type.." when
                receiving Lights list")
        end
    end

    errorResponse = self:getErrorFromResponse(parsedResponse)
    if errorResponse ~= nil then
        error("Could not Fetch lights!!")
        return
    end

    for id,device in pairs(parsedResponse) do
        args = createHTTPDevice(bridgeAddress, device.name, id)
        deviceDiscoveredCallback(args)
    end
end

function Discoverer:createUser(bridgeAddress, restChannel)
    createUserEndpoint = "/api"
    absoluteAddress = bridgeAddress..createUserEndpoint

    createUserMessage = {devicetype = helpers.getApplicationName(),
        username=helpers.getUserName()}
```

```
    messageBody = json.encode(createUserMessage)

    setupText = "Please press the button on the Hue bridge device
       within 30 seconds and press OK to finalize set up."
    showModalDialog("Initializing Hue Bridge", setupText, "OK")
    response = restChannel:Post(absoluteAddress, messageBody)

    errorResponse =
       self:getErrorFromResponse(json.decode(response.Content))
    if response.StatusCode ~= 200 or errorResponse ~= nil then
        error("Could not create user! Got Status Code
           "..response.StatusCode)
    end
end

function Discoverer:getErrorFromResponse(response)
    return response[1] and response[1].error or nil
end

function Discoverer:getLightsFromBridge(bridgeAddress, restChannel)
    getLightsEndpoint = "/api/"..helpers.getUserName().."/lights"
    absoluteAddress = bridgeAddress..getLightsEndpoint

    return restChannel:Get(absoluteAddress)
end
```

A.3 Controls

```
require "controls"
json = require "json"
require "HueHelpers"

HueDimmer = DimmerControl:subclass("HueDimmer")

function HueDimmer:init(args)
    self.super:init(args)
    self.lastUpdated = os.clock()

    if self.label == "Brightness" then
        self.argName = "bri"
    elseif self.label == "Hue" then
        self.argName = "hue"
    elseif self.label == "Saturation" then
        self.argName = "sat"
    elseif self.label == "Temperature" then
        self.argName = "ct"
    else
        error("Unknown HueDimmer type "..self.label)
```

```
    end
end

function HueDimmer:handleUserUpdate(value)
    self.super:handleUserUpdate(value)
    -- only handle updates every 0.1 seconds at most. It's recommended
       that only 10 calls per second
    -- go to the Hue bridge, which we cant really measure but this
       ought to suffice.
    if os.clock() - self.lastUpdated < 0.1 then
        return
    end
    self.lastUpdated = os.clock()
    self.value = value

    address = self.device.Address
    lightid = self.device.ID

    endpoint =
       address.."/api/"..helpers.getUserName().."/lights/"..lightid.."/state"
    updateMessage = {[self.argName]=value, transitiontime=10}

    getCommunicationChannel("REST"):PutAsync(endpoint,
       json.encode(updateMessage))
end

function HueDimmer:handleDeviceUpdate(value)
    self.super:handleDeviceUpdate(value)
    self.value = value

    address = self.device.Address
    lightid = self.device.ID

    OnControlUpdated(self)
end


HueSwitch = PowerControl:subclass("HueSwitch")

function HueSwitch:init(args)
    self.super:init(args)

    if self.label == "Power" then
        self.argName = "on"
    elseif self.label == "Color Loop Effect" then
        self.argName = "effect"
    else
        error("Unknown HueSwitch type "..self.label)
```

```
    end
end

function HueSwitch:handleUserUpdate(value)
    self.super:handleUserUpdate(value)

    address = self.device.Address
    lightid = self.device.ID

    if value == true then
        value = self.onValue
    else
        value = self.offValue
    end

    endpoint =
       address.."/api/"..helpers.getUserName().."/lights/"..lightid.."/state"
    updateMessage = {[self.argName]=value,
                     transitiontime=0}

    getCommunicationChannel("REST"):PutAsync(endpoint,
       json.encode(updateMessage))
end

function HueSwitch:handleDeviceUpdate(value)
    self.super:handleDeviceUpdate(value)
    self.value = value

    address = self.device.Address
    lightid = self.device.ID

    OnControlUpdated(self)
end
```

References

Apple (2015). Sample Apple Developer Program Requirements. In *Apple Developer Program Information, Rev. 8/12/2015*.

Bluetooth SIG (2016). *Characteristics*. `https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx`. Accessed: 2016-05-21.

Datta, S. K., Bonnet, C., and Nikaein, N. (2014a). CCT: Connect and Control Things. In *9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. pp. 21–24.

Datta, S. K., Bonnet, C., and Nikaein, N. (2014b). An IoT gateway centric architecture to provide novel M2M services. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE, pp. 514–519.

Eclipse Foundation (2016a). *Eclipse SmartHome Concepts: Items*. `http://www.eclipse.org/smarthome/documentation/concepts/items.html`. Accessed: 2016-04-03.

Eclipse Foundation (2016b). *Interface Command*. `http://www.eclipse.org/smarthome/documentation/javadoc/org/eclipse/smarthome/core/types/Command.html`. Accessed: 2016-04-02.

Eclipse Foundation (2016c). *Thing Definition, Channel Categories*. `http://www.eclipse.org/smarthome/documentation/development/bindings/thing-definition.html#channel-categories`. Accessed: 2016-05-21.

Hadim, S. and Mohamed, N. (2006). Middleware: Middleware challenges and approaches for wireless sensor networks. In *IEEE distributed systems online*, , no. 3, p. 1.

Home Assistant (2016). *Components*. `https://home-assistant.io/components`. Accessed: 2016-07-10.

Mottola, L. and Picco, G. P. (2006). Logical neighborhoods: A programming abstraction for wireless sensor networks. In *Distributed Computing in Sensor Systems*. Springer, pp. 150–168.

Mottola, L. and Picco, G. P. (2011). Programming wireless sensor networks: Fundamental concepts and state of the art. In *ACM Computing Surveys (CSUR)*, vol. 43, no. 3, p. 19.

Papadopoulos, N., et al. (2009). A connected home platform and development framework for smart home control applications. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*. IEEE, pp. 402–409.

Philips (2016). Hue System Performance. *Hue Developer Program*. `http://www.developers.meethue.com/documentation/hue-system-performance`. Accessed: 2016-05-12.

Presser, A., et al. (2008). Upnp device architecture 1.1. In *UPnP Forum*, vol. 22.

Riliskis, L., Hong, J., and Levis, P. (2015). Ravel: Programming iot applications as distributed models, views, and controllers. In *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*. ACM, pp. 1–6.

Shelby, Z. and Chauvenet, C. (2012). The IPSO Application Framework draft-ipso-app-framework-04. In *IPSO Alliance, Interop Committee*.

Shelby, Z., et al. (2016). *Media Types for Sensor Markup Language (SenML)*. Internet-Draft draft-jennings-core-senml-05, Internet Engineering Task Force. `https://tools.ietf.org/html/draft-jennings-core-senml-05`. Work in Progress.

Townsend, K., et al. (2014). *Getting started with Bluetooth low energy: Tools and techniques for low-power networking*. " O'Reilly Media, Inc.".

Zhu, Q., et al. (2010). Iot gateway: Bridgingwireless sensor networks into internet of things. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*. IEEE, pp. 347–352.