Developing Real life, Task Oriented Applications, For The Internet of Things

[1]Ali Ali Hussein Atef, [2] Bhojraj Agarwal
[1]M.Tech Scholar, [2],Assistant Professor (ECE Department)
Vivekanada Global University, Jaipur.

Abstract : The Internet of Things is becoming ubiquitous. A growing number of objects are being equipped with devices that interact with its environment and exchange data via the Internet. Such devices are not powerful enough to run iTasks (an implementation of the Task Oriented Programming paradigm written in Clean) applications. The mTask Embedded Domain Specific Language was created with one goal in mind: to bring Internet of Things devices and the Task Oriented Pro-gramming paradigm together. But so far, only trivial applications were developed using mTask.

***IndexTerms* – Internet of Things (IOT), mTask, Application.**

I. Introduction

The Internet of Things (IoT) consists of a network of “things” (devices, computers, systems, etc.) that interact with each other via the Internet. These components can exchange data, monitor and manage each other. IoT is a global, growing phenomenon. According to Gartner, there were 3.96 billion connected “things” in 2016 and by 2020, 12.863 billion devices are expected to be connected to the Internet IoT has been used in a myriad of applications including home automation, fitness tracking, health care, warehouse monitoring, agriculture and industry manufacturing. IoT devices can be dedicated servers, personal computers, tablets, smartphones, smartwatches or compact devices operated by microcontrollers. Microcontrollers are small, cheap computers with limited resources and low power consumption commonly used to interface with the real world. They often gather data from sensors (movement, light, temperature, etc.), act on actuators (motors, LEDs, switches, etc.) and communicate with other devices. Task Oriented Programming (TOP) is a new programming paradigm used to develop online, collaborative applications. Its central concept, a task, can be used to model different types of work performed both by users and systems. TOP provides a high level of abstraction, liberating the programmer from the burden of technical details, such as user interfaces. The iTasks system implements TOP in the functional programming language Clean as an Embedded Domain Specific Language (EDSL). It automatically generates as many assets as possible, turning it into a great tool for rapid prototyping. Given an iTasks program, the system automatically generates a web application that can be accessed via a web browser. Users can access this application to inspect and work on tasks. The system has been proved useful in many fields, including incident response operations and navy vessels automation.

Although iTasks applications often require user interaction, some tasks could be automated. Examples are tasks that interact with the external world: reading room temperature, blinking an LED, detecting movement, unlocking a door, etc. The iTasks environment could benefit from such automation. Microcontrollers pose as great candidates to interface with the external world. They are affordable, energy efficient and are seamlessly combined with sensors and actuators. Unfortunately, — due to hardware limitations — microcontrollers are not suitable to run iTasks tasks. To bridge this gap, the mTask Domain Specific Language (DSL) was created to enable the execution of simple tasks on microcontrollers, bringing such devices to the iTasks world.

Even though mTask was created to allow iTasks tasks to run on IoT devices, it has not been proved capable of running real-life applications yet. The examples built during its development were simple demonstrations and were far from real-life IoT applications. Given that, I propose the following research question:

Is it possible to develop real-life, IoT applications using mTask? If so, how can the development process be improved? If not, what are the challenges to solve to make it possible?

I plan to tackle the research question by example. Namely, trying to develop a real life IoT application using mTask. The attempt to develop such an application should display mTask’s capability to create real life applications while displaying new opportunities to improve the development process.

The application I chose to develop tackles a popular problem in IoT: home automation. This application would be responsible for automating simple home management tasks such as turning the central heating system off when the room is warm, or opening up the curtains at a set time. The proposed application requires several IoT devices equipped with sensors (temperature, light, humidity, etc.) and actuators (LEDs, motors, relays, etc.) spread across rooms.

II. Embedded Domain Specific Languages

A General Purpose Language (GPL) is a computer language intended to be used in a wide range of domains. An example is the C++ programming language, which can be used in domains that vary from video games to web servers and systems programming. In contrast, a Domain Specific Language (DSL) is a computer language that was designed to be used in a particular domain. Game Maker Language, HTML, LaTeX and VHDL are examples of DSLs. These languages, when compared to GPLs, offer a higher level of abstraction from their target domain.

II.I Deep Embedding

A deeply EDSL is represented in its host language as an Algebraic Data Type (ADT), where language constructs are modelled as data constructs. Views are functions that take the ADT as input and return another ADT that represents its semantics. An example of a simple deeply EDSL and its views (pretty printing and evaluation) can be seen on Listing 2.1.

```
:: MyDSL = I Int | B Bool
| Add MyDSL MyDSL
| Sub MyDSL MyDSL
| And MyDSL MyDSL
| Or MyDSL MyDSL
| Var String prettyPrint ::          MyDSL eval :: ‹MyDSL Int [String]
```

Listing 2.1: A simple deeply EDSL and its views

The biggest advantage of deep embedding is that adding a view to the DSL is easy: simply create a function that transforms the ADT. One disadvantage is that extending the DSL might require a lot of work, since new code has to be created for every new construct in all the views. Another disadvantage is its lack of static type safety. As seen on Listing 2.1, MyDSL allows operations on mixed data types, such as addition on booleans and disjunction on integers.

II.II Shallow Embedding

Building a shallowly EDSL consists of representing the language constructs directly as its se- mantics.

III. The Application

The application developed during the research should ideally display characteristics inherent to IoT applications. Although, some of there characteristics energy consumption, performance and security — were not considered during the research. These aspects were ignored because the development of iTasks did not take them into consideration. Additionally, some criteria were added based on the research question.

The following criteria were used to choose an application:

Suitable The application should solve a problem that is suitable for mTask. This narrows the choice to IoT applications that can be developed for platforms supported by mTask.

Non-trivial Trivial applications (e.g. an LED blinking) were previously developed. There- fore, the application should not solve a trivial problem — e.g. a simple hallway motion-activated light sensor. It should go beyond a purely reactive system. It does not have to solve a novel problem, but its development should be adequately challenging.

Simple Due to time constraints, it should be simple enough to be developed during this re- search. Since building a full-pledged application is not the goal of the research, some concerns as feature completeness, user experience design and security are not taken into account. Its source code should not be too complex.

Interesting It is not enough that the application is suitable and technically good. It should tackle an existing, interesting problem. Ideally, a problem which users inserted into the applica- tion domain would be willing to pay for a solution.

Significant The application should somehow improve the environment it is inserted into. Ex- amples are accelerating an assembly line, saving commute time, improving one’s health or well being or reducing operational cost.

Comprehensible Its domain and main features should be easily understandable by non- domain experts. Its functionality details and operational features might require specific know- ledge, but the application should be easily described on a high level to someone who is not familiar with its domain. Comprehensibility is relevant because it improves the application and therefore the research’s reachability.

Robust The application should be able to handle errors to some extent. It should at least be able to detect and communicate device disconnection. Ideally, it would automatically migrate tasks from disconnected devices to available devices whenever possible.

Highly connected It should support multiple devices simultaneously. These devices should be able to exchange information (e.g. sensor values) when suitable. Ideally, the devices would be connected wirelessly.

Dynamic The application should not be static. Given that the interpreted version of mTask (Section 4.2) is being explored, its dynamic nature should be exploited. The application domain should naturally allow dynamicity. Ideally, tasks would be sent to and removed from devices regularly.

Diverse It should use as many peripherals as possible. Since IoT applications often handle a heterogeneous group of peripherals, it is important that a diverse group of sensors and actuators is used. The application should not restrict itself to a couple of peripherals.

Extensive The application should use mTask features extensively. Given that it is testing mTask’s capabilities, it is important that the application tests as many mTask features as pos- sible. There is a correlation between the number of features used by the application and the certainty about mTask’s abilities.

IV. Application Development

After the application domain was chosen, development began. Although Autohouse was de- veloped during the research, it was not the research object. It was used merely as a tool to assess mTask’s capabilities and thus answer the research question. Therefore, this chapter will not focus on Autohouse itself, but on the limitations of mTask unearthed during its development. A quick overview of its development is given in Section 6.1. Some limitations of mTask were overcome and are described in Section 6.2. Other limitations remain and are described in Sec- tion 6.3. Finally, Section 6.4 describes how automatic task migration was accomplished without modifying mTask.

Development Overview

Autohouse is an application developed using Clean, iTasks and mTask. Due to time constraints, it was thoroughly tested only on macOS 10.13 (High Sierra). It was tested on Linux (Ubuntu 16.10) on early stages of development. Autohouse’s source code is available at its GitHub repository1.

Application Architecture

Autohouse development started just like many iTasks applications: defining ADTs and SDSs. The application has ADTs that model key concepts in the home automation domain: House (representing the smarthome), Room and Unit (representing a device). A House is a list of Rooms and a Room is essentially a list of Units.

All the application data lives in one SDS that represents the entire house. Other SDSs (e.g. for rooms and units) are derived from this main SDS using parametric lenses. Rooms and units have unique ids that are used to locate them in the house SDS. The default automationtasks are stored in a list of Program — in Autohouse’s source code, automation tasks are named programs to avoid name clashing with iTasks’ Task ADT. The Program record contains all the information inherent to an automation task.

The application contains three main tasks running in parallel: manage house, manage units and send new task. The first one lets the user create and edit rooms. The second allows the user to inspect, send tasks to and disconnect units. The last task lets the user pick a mTask task to send to a device that is compatible with it.

Once the iTasks foundation was created, the mTask development could begin.

1Autohouse on GitHub. Available at

https://github.com/matheusamazonas/autohouse.

Using the Simulator

As expected, device features were first implemented using simulators (Section 4.2.4). These devices proved to be great for early stages of development. First, multiple simulators can be easily instantiated, which was particularly useful when physical devices were not available yet. Simulators allowed the development of peripheral tasks even before some peripherals were avail- able for testing. Additionally, given that simulators are highly customizable, some features that depended on different device configurations were tested. Such freedom to quickly choose between different device configurations does not exist when dealing with physical devices.

Although the simulator was particularly helpful during early stages of development, it proved to be a great debug tool during later stages as well. Whenever a device behaved unexpectedly, the same environment was reproduced using a simulator. Then, using the debugging UI, the device's state (i.e. memory, tasks, program counter, peripheral values) could be inspected. Using the simulator as a debug tool became standard practice during Autohouse's development.

Device Communication

Once prototyping using the simulator was over, development moved to physical devices. But one problem had to be solved before deploying mTask tasks on Arduinos: wireless device communic- ation. Ideally, Autohouse would communicate with its devices wirelessly. But so far, only Serial connections over USB were used to connect to Arduinos. Two wireless options were taken into consideration: Wi-Fi (through TCP) and Bluetooth (through Serial).

The first option tested was Wi-Fi, specifically with the ESP8266. The ESP8266 is a system on a chip with a microcontroller, a full TCP/IP stack and Wi-Fi. It can be used as a Wi-Fi module (giving other microcontrollers Wi-Fi capabilities) and as a standalone microcontroller. In Autohouse's setting, it would be used to give the Arduino boards Wi-Fi capabilities. A microcontroller can use Hayes commands2 to control the ESP8266 as a Wi-Fi module. Thus, a library was required to interface with it. A couple of open-source libraries were tested3,4, but they either did not offer some necessary features (e.g. retrieve the list of connected clients) or behaved unexpectedly (e.g. losing received messages). After some unsuccessful attempts to adapt the existing libraries, Wi-Fi was put aside and Bluetooth was considered as an alternative.

The Bluetooth module tested was the HC-05, a Bluetooth 2.0 Serial Port Protocol (SPP) module. Since this module runs through Serial, it can be directly connected to the Arduino board's Serial pins (TX and RX). Incoming data is preprocessed by the HC-05 and send directly to the device's Serial port. Therefore, no client data processing is required to transmit data via Bluetooth. Additionally, no code changes were necessary to support Bluetooth connection. When compared to Wi-Fi, Bluetooth brought clear advantages — i.e. it works out of the box and does not required additional code — and therefore was chosen to be used as Autohouse's wireless solution.

Device Deployment

Once the wireless communication was set up, development moved to physical devices. First, all peripherals were tested using a single Arduino board to ensure that they were compatible. Once

2Hayes command set - Wikipedia. Available at https://en.wikipedia.org/wiki/Hayes_command_set. Ac- cessed on September 14th 2018.

3WiFiEsp on GitHub. Available at https://github.com/bportaluri/WiFiEsp/tree/master/src. Accessed on September 14th 2018.

4ESP8266 wifi on GitHub. Available at https://github.com/ekstrand/ESP8266wifi. Accessed on September 14th 2018.

all peripherals were tested using mTask tasks, more devices were added to the Autohouse system. The test system contained five Arduino Uno compatible boards. Each board was equipped with a HC-05 Bluetooth module, two LEDs, two push down buttons, a PIR and temperature, humidity, ultrasonic and analog light sensors. The boards had similar capabilities in order to test task migration (tasks should only migrate to devices that are compatible with it). Only one board was equipped with a servo. This choice was also motivated by the task migration feature: if a device running a task that uses a servo disconnects from the server, its task should not migrate because no other device has a servo.

Once device deployment finished, Autohouse's standard task list was created and tested. Autohouse's version5 0.1.0 corresponds to the end of this development phase.

Changes to mTask

Limitations of mTask surfaced during the development of Autohouse. Some of these limitations were overcome by changing mTask. These changes are described below.

Variables

Since mTask is an imperative language, it would benefit from mutable data features. Although there are no mTask constructs to represent variables, SDSs might be used as updatable data containers. In such a setting, an SDS is created for each desired variable. This trick brings updatable data storage to mTask, but it prompts two problems.

First, there is no separation of concerns. Variables and SDSs should be, by definition, different things. A variable is a local updatable data storage in memory. An SDS is an abstraction layer over any kind of shared data, including data in memory. Using an SDS locally goes against what a shared data source represents. Second, when SDSs are sent to devices, they are not attached to a specific task. Also, on the current version of mTask, there is no way to establish whether an SDS belongs to a given task. As a consequence, SDSs are never deleted from devices. Variables, on the other hand, are always bond to a specific task and could be removed with their correspondent task altogether, saving space in the device's memory. Thus, mTask could benefit from a language construct for variables.

The vari class was created to fill this gap. It contains two functions: vari and con, representing variable and constant data storage respectively. Its definition can be seen in Listing 6.1. From a language construct point of view, the sds and vari classes do not differ much. Both classes contain constructs that might be used as updatables and as expressions. But there are two differences between these classes. First, vari contains a construct for constant data: con. Second, vari functions expect a value of type t as its initial value (seen as the first argument of in Listing 6.1). The sds function expects a (Shared t) instead. The biggest difference between the sds and vari classes regards their behavior on the interpreted view of mTask. Vari- ables belong to a task and will live as long as the task lives. SDSs are not bound to a task and will live in the device indefinitely.

```
:: Vari  = Vari
instance is Expr Vari
instance is Upd Vari
```

5Autohouse release 0.1.0 on GitHub. Available at https://github.com/matheusamazonas/autohouse/ releases/tag/0.1.0.

```
class vari v where
vari :: ((v t Vari)         In t (Main (v c s)))          (Main (v c s))
con ::  ((v t Expr)         In t (Main (v c s)))<         (Main (v c s))<
```

Listing 6.1: The vari class

Listing 6.2 displays an example of variables in mTask: the task blink. This task blinks LED1 based on the value of variable v. The variable v is created using the vari construct. Its value is updated using the =. infix operator, similarly to SDSs. It can also be used as a boolean expression, as the condition to an IF construct.

All Autohouse automation tasks must be of type Main (v () Stmt). The noOp after the attribution in Listing 6.2 is required to ensure that the program matches that type. The noOp construct is a wild card used whenever the type of a construct does not match its desired type.

```
blink :: Main (v () Stmt) | program v blink = vari λv = False In { main =IF (v)
(ledOn (lit LED1)) (ledOff (lit LED1)) :.v =. Not v :. noOp}
class noOp v where noOp :: v t p
```

Listing 6.2: Example of the usage of variables in mTask

The addition of variables to the language required changes on mTask’s communication pro- tocol (Section 4.2.2). When a task is sent to a device, its variables must be sent as well. Therefore, a MTTask message must include the variables used by the given task. Variables are modelled in the BCVariable record. A variable contains a unique (within a task) identifier and its initial value. The BCVariable record and the communication protocol change can be seen in Listing 6.3.

```
:: BCVariable = { vid :: Int, vval :: BCValue }
:: MTaskMSGSend
= MTTask Int MTaskInterval [BCVariable] String...
```

Listing 6.3: Change in mTask’s communication protocol to accommodate task variables

Additionally, the simulator and the client engine were modified to support task variables. When a task is received, its variables are stored. During task execution, variables are fetched and assigned similarly to SDSs. When a task terminates, its variables are removed from the device.

Peripheral Code

The mTask library already supported some of the peripherals Autohouse planned to use: LEDs, analog and digital pins. Although, new peripherals (e.g. light, temperature and humidity sensors) were required by some of the default automation tasks. Following the natural development process of an mTask application, these peripherals were first emulated using the simulator.

As more peripherals were implemented, it was clear that the workflow required to add a new peripheral to the system could be improved.

Adding a new peripheral required changes on different parts of mTask. An overview of the necessary changes can be seen below.

A new class that represents the peripheral is added to the language.

Depending on the peripheral, a new ADT is created to represent its values (e.g. DigitalPin).

New bytecode instructions are created.

Bytecode encodings are updated to support the new instruction and the possibly new ADT.

The MTaskDeviceSpec record is modified to include a flag for the new peripheral.

The simulator interpreter is updated to handle new bytecode instructions.

The C client is modified to handle the new peripheral.

The changes on the C client code depended heavily on the type of peripheral being imple- mented. Changes on the clean code though, were often similar. Previously, peripheral code was scattered around the mTask library. Peripheral classes were inside the Language module along with possibly new ADTs. Instances of the peripheral classes for each mTask view were in the re- spective view’s module. The simulator interpreter contained peripheral-specific code. Bytecode encodings for basic types were mixed with encodings for peripheral data types. Overall, adding a new peripheral was particularly cumbersome and extremely error-prone. Finally, there was no separation of concerns whatsoever.

A new modular code architecture for peripherals was introduced to solve the problems de- scribed above. It aims to remove peripheral-specific code from mTask core and simulator modules. In this architecture, each peripheral should be defined in its own module. Its type class, ADTs, bytecode encodings and view instances are defined in that same module. The simulator does not have any peripheral-specific code. Instead of explicit fields for each peripheral, the simulator state record (SimState) contains a list of Peripheral. This new data type is a wrapper around every mTask peripheral. Its definition can be seen in Listing 6.4.

```
:: Peripheral = E.e: Peripheral e & peripheral e
class peripheral e | iTask e where
processInst :: BC e       State SimState (e,Bool)
```

Listing 6.4: The Peripheral class

The peripheral class was created to enable the removal of peripheral-specific code from the simulator interpreter. Its only function, processInst defines how a peripheral should interpret bytecode instructions (BC). Naturally, a peripheral should only interpret instructions that are relevant to it. The simulator interpreter executes one instruction at a time. If an instruction be- longs to mTask’s core instruction set (excluding peripheral instructions), the interpreter executes it immediately. If the instruction does not belong to the core instruction set, it is assumed to be a peripheral instruction and it is presented to all simulator peripherals using the processInst function. Once a peripheral responds to an instruction (represented by the Bool on processInst returned value), the interpreter considers the instruction executed and stops looking for a peri- pheral to execute it. If no peripheral executes the instruction, an error (”instruction unknown”) is thrown.

The addition of new bytecode instructions remains outside of the peripheral modules. Al- though technically it is possible to extend the bytecode data type (BC) across separate modules, the amount of work necessary to do so outweighs the benefits it could bring. Currently, BC’s in- stance of the iTask type class is automatically derived. Clean can not automatically derive type classes of extended types. Therefore, if BC was extended, an instance of iTasks type class would have to be manually derived. Doing so would bring peripheral-specific code back to language core modules, going against the intend that drove the change to begin with.

The development that followed the changes described above proved that the separation of concerns regarding peripheral code improved mTask. Peripherals were added faster, with less code changes and less errors. Additionally, code maintainability increased substantially. Since peripheral code lays mostly in the same module, small changes can be performed faster and safer.

New Peripherals

Previously, the interpreted mTask supported the following peripherals: LEDs, LCD displays (for displaying numbers only), analog and digital pins. The standard Autohouse tasks required new peripheral support. The following peripherals were added to mTask: DHT22 temperature and humidity sensor, HCSR04 ultrasonic sensor, digital light sensor, Grove analog light sensor (P) V1.1, PIR and servo.

The digital light sensor, the Grove analog light sensor and the PIR did not require an external library to be used. Their data pin is connected to board pins and their values can be read using Arduino standard functions. An additional library was required to interface with the servo. The Arduino Servo library6 was used. An additional library was also required to interface with the

DHT22 sensor. Although there are libraries available out there, I decided to implement a small and simple one just for mTask: DHTino7. This choice was motivated by the limited amount of flash memory (32 KB) on the Arduino Uno. Existing libraries support many different sensors and have many features that would not be used by mTask. As a consequence, these libraries would take too much flash memory space. Guided by the same motivation, the Ultrino8 library was created to interface with the HCSR04 ultrasonic sensor.

Device Requirements

Some tasks rely on certain peripherals to execute. For example, a task that regulates room temperature relies on a temperature sensor. Despite that, mTask did not provide a mechanism to determine whether a task is compatible with a device. The Requirements view was created to bring this feature to mTask. Its definition can be seen in Listing 6.5. Requirement is a type constructor with two phantom type variables: a and b. These type variables are required by mTask type classes. Requirement is a wrapper around the device specification type MTaskDeviceSpec.

Given a mTask construct, this view will return the minimum device specification necessary to support that construct. This information can be used to determine whether a device matches the minimum specification for a task and therefore, if it is compatible with it. The match function

(seen in Listing 6.5) does exactly that. Given an mTask program and a Maybe MTask Device Spec, it yields whether the device and program are compatible.

6Arduino Servo Library. Available at https://www.arduino.cc/en/reference/servo. Accessed on September 10th 2018.

7DHTino on GitHub. Available at https://github.com/matheusamazonas/DHTino. Accessed on September 10th 2018.

8Ultrino on GitHub. Available at https://github.com/matheusamazonas/Ultrino. Accessed on September 10th 2018.

:: Requirements a b = Req MTask Device Spec match :: (Main (Requirements a b)) MTask Device Spec
‹instance arith Requirements
instance UserLED Requirements
Bool

Listing 6.5: The Requirements view

Instances of mTask classes (including peripheral classes) are defined for Requirement. There- fore, given a mTask task, an application can filter the available devices based on whether they are compatible with it. The opposite is also possible: given a device, an application can filter tasks based on whether they are compatible with it.

Device Disconnection

By design, Autohouse should be robust regarding device disconnection (Section 5.4). Ideally, the system would detect a device disconnection and migrate the device’s tasks to another suitable device. There were two challenges to tackle in order to implement this feature.

First, mTask does not recognize device disconnection for all of the device types it supports. Simulators never get disconnected. TCP devices throw an iTasks error when a disconnection is identified. This error is not caught by mTask and propagates upwards. Serial devices kill the application when disconnected. The library used by mTask to connect to Serial devices (CleanSerial9) halts execution when a device is disconnected.

In order to detect device disconnection, mTask had to be modified. If the device commu- nication fails, the channelSync task (Section 4.2.5) should throw an exception10. TCP devices already throw an exception when communication fails and therefore require no change. Although simulators never disconnect from the system, simulating a disconnection would benefit testing. Hence, simulators were modified to support intentional disconnection. CleanSerial was modified to support device disconnection recognition.

Second, even if mTask recognizes device disconnection, it still cannot communicate it to Autohouse. Ideally, mTask would communicate device disconnection through an error handler that would be provided by the application. Thus, the application would decide what task to perform in case of a disconnection. As seen in Section 4.2.5, the mTask library provides a single function to connect with a device: with Device. This function is responsible (besides other tasks) to manage the connection to the device and therefore was the perfect place to insert an exception handler. An exception handler is a task that takes an error String as input. Listing 6.6 displays the type signature of the original with Device along with its new version, named with Device’ here. If the application using mTask does not want to handle connection errors, the iTasks throw function can be used as the error handler. Doing so would propagate the exception upwards, emulating the behavior of with Device.

with Device :: a (MTask DeviceTask b) Task b | channelSync a with Device’ :: a (MTaskDevice ‹ Task b) (String ‹ Task ()) ‹ Task b | channelSync a

‹

9CleanSerial on GitLab. Available at git @ gitlab . science. ru. nl: mlubbers/ Clean Serial.git. Accessed on Semtember 8th 2018

10An iTasks task yields either a value or an exception. The iTasks standard library provides functions to create and handle exceptions.

throw :: e Task a | iTask a & iTask e & to String e Listing 6.6: Change in mTask to upport a device disconnection handler

Consequently, mTask recognizes and provides an exception handler for device disconnection. Autohouse uses this feature to detect unit disconnection and thus automatically migrate tasks from the disconnected device to a suitable one.

Simulator Improvements

The simulator (Section 4.2.4) proved to be an essential tool during the development of Autohouse. Although, it was clear that it could be improved to ease debugging and testing of the application. Sometimes, the developer might want to debug a task and inspect it closely. The simulator's manual mode is adequate for such usage, but it might be a bit cumbersome to use. Specially with large tasks, stepping over each program instruction becomes a rather tedious and inefficient process. With that in mind, the simulator was extended to support breakpoints on bytecode instructions. Tools to add and to step over breakpoints were added to the simulator UI. When executing a task, the simulator goes through its bytecode instructions, checking if there are breakpoints on each instruction before executing it. If an instruction has a breakpoint, execution waits for user input (by clicking on "step over") to continue. At any point, the user is able to edit breakpoints.

The ability to simulate peripheral values is crucial for program testing in mTask. Tasks often rely on peripheral values and therefore can only be thoroughly tested if peripheral values can be simulated. Although, the simulator did not have such feature. The development of Autohouse showed how necessary this feature is for mTask development. Hence, simulation of peripheral values was incorporated into the simulator. Values can be manually set via the simulator UI, similarly to breakpoints.

Limitations of mTask

Some of the limitations of mTask that surfaced during the development of Autohouse were not overcome. First, SDSs can not be removed from a device. There is no message in mTask's communication protocol (Section 4.2.2) to request SDS deletion. Since an SDS is not bound to a task, once it is sent to a device, it lives there indefinitely. As a consequence, a device can accumulate unused SDSs over time, possibly filling the device's memory with dangling SDSs. Ideally, SDSs would always be bound to a task. Thus, they would be removed along with its task, eliminating dangling SDSs.

Second, there is a communication loop on SDS updates. The mTask library automatically synchronizes SDSs between server and devices. Whenever a device publishes an SDS value, a message is sent to the server, which updates the actual SDS. Also, the server observes an SDS and whenever it is modified, it sends an update message to every device that uses that SDS. Hence, the server ensures SDS synchronization. A problem arises because the server is not aware of who is updating an SDS. Therefore, it might send an update message to the same device that published the SDS value, creating a communication loop. This behavior is not desirable because it generates unnecessary communication between the server and the devices. Additionally, it might create unexpected behavior. Ideally, the server would identify who is updating the SDS and avoid sending an update message to the device that trigerred it.

Additionally, mTask does not communicate whether a task sent to a device was acknowledged. Ideally, a call to liftmTask would communicate task acknowledgment using a callback handler (similarly to the connection error handler in Section 6.2.5). The application might want to wait for the task acknowledgment to continue. For example, Autohouse could wait for a task acknow- ledgment to store the task data in the unit SDS. Since it can not detect task acknowledgment, it stores the task data before actually sending the task. As a consequence, if a task fails to be acknowledged, an invalid task will live in the device's task list.

Finally, mTask does not support floating-point arithmetic. Some sensors data (e.g. DHT22 temperature and humidity sensor) is represented using floating-points and could not be directly translated into mTask bytecode values. As a workaround, the Arduino client uses integers to represent floating-points, with a precision of two decimals.

Task Migration

In case of device disconnection, the application should migrate the unit's tasks to another suit- able one. The mTask library offers means to connect and send task to devices, but not to manage them. The application using mTask is responsible for device management. Therefore, automatic task migration was implemented entirely in Auto house and required no changes to mTask whatsoever.

First, the feature behavior was defined. Although automatic task migration might be helpful, not all tasks should be automatically migrated. For example, some tasks that are suitable for a hallway unit (e.g. motion-activated light switch) might not be suitable for a bedroom unit. Therefore, different migration strategies were created. These strategies are represented in the Migration ADT, seen in Listing 6.7.

:: Migration = DoNotMigrate | SameRoom | AnyRoom

Listing 6.7: Task migration strategies of Autohouse

Tasks with Do Not Migrate migration strategy never migrate to another unit. Tasks with the Same Room strategy migrate only to units within the same room of its original unit. Tasks with Any Room strategy migrate to any other unit in the smart home. When a task is being migrated, other units are checked for compatibility (based on the task's strategy, following no particular order) and once a compatible unit is found, the task is sent to it. If no compatible unit is found, the task is not migrated. The Auto house user is responsible for determining a task's migration strategy when sending it to a unit.

The application has to save enough task data to migrate a task in case of a device disconnection. The mTask's liftm Task function (Section 4.2.5) is used to send tasks to devices. Besides the device itself, this function takes a task interval and an mTask task as input. Therefore, this is all the information the application needs to send a task to a device. Auto house's default tasks have a unique id that can be used to retrieve the task from the default task list. Thus, the application should only need to store the task index and its interval (MTaskInterval) in order to migrate it.

But storing the task id and interval is not enough. Some Auto house tasks require user- provided arguments to work. For example, a user must provide an initial target temperature to a thermostat task before sending it to a device. If such a task is being migrated, the application should be able to restore the task arguments as well. Auto house stores task arguments along with the task index and interval. All the information necessary to migrate an Auto house program lays in the Program Instance data type, seen on Listing 6.8.

:: ProgramInstance = { pIx :: Int, pArgs :: [Dynamic], pInt :: MTaskInterval, pMig :: Migration }

Listing 6.8: Task migration data

A unit contains a list of Program Instances. Each list element represents one task running on the unit and can be used to migrate its respective task to a new device. If a unit disconnects, the application goes through the unit's Program Instance list and migrates the tasks accordingly. Automatic task migration was tested using simulators and physical devices and it worked as expected.

mTask

Recent research has been conducted on mTask. A new, task-based version of it has been pro- posed. On this version, the imperative language has been replaced by a functional one. This new version was not used during the research reported in this document because its implementation was not available on time.

The usage of programming languages to interface with microcontrollers has been a subject of research. Firmata1 is a protocol to control microcontrollers. Its messages follow the MIDI message format and model mostly commands on analog and digital input and output pins. There is a client-side implementation for the Arduino2 and host-side implementations for many programming languages, including a Haskell implementation to communicate with Arduinos called hArduino3. Since Firmata is a protocol and not a programming language, full applications cannot be built using Firmata solely. Other tools are built on top of it.

The Haskino library enables Arduino programming using Haskell. The library is available in two different flavors. The first one is based on hArduino (and consequently, on Firmata) and requires the Arduino to maintain a serial connection with the host. On this approach, most of the program evaluation is executed on the host and only I/O commands run on the client. The second approach drops Firmata and uses its own communication protocol. The client is more independent and can execute more elaborate commands, including control flow constructs. In contrast with the first approach, it presents a lower communication overhead. In addition, programs can be written to the Arduino's EEPROM, allowing standalone execution. When compared to mTask, both flavors of Haskino depend heavily on the server.

Some research has been made on generating C/C++ code for microcontrollers from high level languages. Ivory is an EDSL embedded into Haskell that generates safe embedded C code. By design, the generated code is memory safe and free from common errors and undefined behaviors. It uses Haskell's type system (with some GHC extensions) to avoid errors like array indexing out of bounds, main loop function with return statements and dangling pointers. Additionally, it prohibits (by design) some standard C features that might generate unsafe code. Ivory was used on the development of the SMACC Pilot, a high-assurance autopilot system for quadcopter Unmanned Air Vehicles (UAV). Unlike mTask, Ivory does not support

1Firmata Protocol Documentation. Available at https://github.com/firmata/protocol. Accessed on August 10th 2018.

2Firmara Arduino. Available at https://github.com/firmata/arduino. Accessed on August 10th 2018.

3hArduino. Available at http://leventerkok.github.io/hArduino. Accessed on August 10th 2018. dynamic uploading of new tasks.

The frp-arduino library4 implements the Functional Reactive Programming (FRP) paradigm as an EDSL embedded in Haskell. Programs in the EDSL can be compiled to Arduino C code which can be uploaded to Arduino boards. Juniper5 is another FRP language for the Arduino. It is a standalone programming language that transpiles to Arduino C++.

Additionally, some programming language interpreters were ported to microcontrollers. Es- pruino6 is a JavaScript interpreter for microcontrollers. It officially supports only proprietary boards but other microcontrollers such as the ESP8266 and the members of the STM32 family are supported by the community. Due to hardware limitations, none of the Arduino boards are supported. Espruino's official website lists many projects that were built using it, including home automation applications.

Micropython7 is a lean implementation of the Python interpreter and parts of its standard library for microcontrollers. Its main target device is the proprietary pyboard. Given that it requires at least 16KB of RAM, it is not compatible with most Arduino boards. It is compatible with microcontrollers of the STM32 family. Many projects (including home automation) were developed using Micropython and pyboards.

Finally, the programming of microcontrollers dynamically (without the need to plug it to a computer) is a well known practice. For example, the ESP82668 Wi-Fi module supports Over- the-Air (OTA) programming. The Arduino Uno Wi-Fi9 is a version of the Arduino Uno board that contains an ESP8266 module and supports OTA programming natively via the Arduino IDE. It is important to note that although OTA enables dynamic programming of microcontrollers, it differs from mTask's dynamicity. On OTA programming, the device memory is reset when a new program is loaded. On the dynamic version of mTask, the device's memory and the tasks running on it are unaffected.

Autohouse

Autohouse is a home automation application built with the intent to assess mTask's capabilities and was not meant to be commercialized. Therefore, when compared to existing commercial home automation systems, mTask supports less platforms, offers less features and does not focus on some aspects (e.g. security, performance, user experience). Although, there are some fundamental differences between these applications and Auto house. A brief comparison follows. The open HAB10 is an open-source home automation integration platform. Instead of con- trolling devices running a custom firmware, open HAB integrates existing automation system from different manufacturers. Therefore, Auto house and open HAB operate in different abstraction levels.

Home Assistant11 is an open-source automation system that supports many automation plat- forms including Arduino. Users can configure automation tasks using YAML files. Although it

4FRP on Arduino. Available at https://github.com/frp-arduino/frp-arduino. Accessed on August 10th 2018.

5Juniper Programming Language. Available at http://www.juniper-lang.org/index.html. Accessed on Au- gust 11th 2018.

6Espruino. Available at https://www.espruino.com. Accessed on August 10th 2018.

7Micropytohn. Available at https://micropython.org. Accessed on August 10th 2018.

8ESP8266 Overview. Available at https://www.espressif.com/en/products/hardware/esp8266ex/overview.

Accessed at August 11th 2018.

9Arduino Store - Arduino Uno Wi-Fi. Available at https://store.arduino.cc/arduino-uno-wifi. Accessed on August 11th 2018.

10openHAB. Available at https://www.openhab.org. Accessed on September 19th 2018.

11Home Assistant. Available at https://www.home-assistant.io. Accessed on September 19th 2018.

supports Arduino, the server uses Firmata protocol (Chapter 7) to control the devices. Therefore, evaluation of automation tasks is performed on the server, not on the clients. As a consequence, if the connection between server and clients fails, the devices stop performing tasks. In Auto house, devices keep executing tasks if communication is interrupted.

Blynk12 and Thinger.io 13 are IoT platforms in which devices (e.g. Arduino, Mbed, Raspberry Pi14) can be controlled via iOS and Android apps. Unlike in Autohouse, new tasks cannot be sent to the devices dynamically.

As seen above, mTask's microcontroller support and dynamic nature brings a set of features to Auto house that none of the analyzed home automation systems possesses.

12Blynk. Available at https://www.blynk.cc. Accessed on September 19th 2018.

13Thinger.io. Available at https://thinger.io. Accessed on September 19th 2018.

14Raspberry Pi. Available at https://www.raspberrypi.org. Accessed on September 19th 2018.

Conclusion

The research reported in this documented tested mTask's ability to develop real-life IoT applications. The research question was tackled by example: the Autohouse application intended to assess mTask's capabilities. The application is a home automation system that allows users to dynamically manage automation tasks running on devices spread across different rooms.

Limitations of mTask surfaced during the development of Autohouse. Some limitations were overcome by changing the mTask and CleanSerial libraries. Task variables were added to the language. Device disconnection recognition was implemented, allowing the application to auto- matically migrate tasks when a device is lost. A new view was added to the EDSL which generates minimum device requirements for a mTask task. This view can be used to filter available devices based on whether they support a given task. Six new peripherals were added to the mTask language and to the Arduino client. Peripheral code was restructured, easing the addition of new peripherals, increasing code maintainability and bringing a better separation of concerns between the language core constructs and peripheral constructs. Finally, the simulator for the interpreted mTask was modified to support the setting of peripheral values and breakpoints, which improved testing and debugging considerably.

Other limitations could not be overcome during this research. SDSs are never removed from devices and live there indefinitely. There is an unwanted communication loop between devices and server whenever a device publishes an SDS. The mTask library does not communicate task acknowledgment. Although these limitations were not overcome, they did not stop the development of Autohouse.

The mTask EDSL and library were successfully used to develop a real-life IoT application: the home automation system Autohouse. Some of the limitations unearthed during the development process were overcome and some remain.

References

[1] P. Achten, P. Koopman and R. Plasmeijer. "An Introduction to Task Oriented Program- ming". In: Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers. Ed. by V. Zs´ok,

[2] Z. Horv´ath and L. Csat´o. Cham, Switzerland: Springer International Publishing, 2015, pp. 187–245. Arduino - Home. url: https://www.arduino.cc (visited on 06/08/2018).

[3] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer and M. J. Plasmeijer. "Clean — A language for functional graph rewriting". In: Functional Programming Languages and Computer Architecture. Ed. by G. Kahn. Berlin, Heidelberg: Springer, 1987, pp. 364–384.

[4] J. Carette, O. Kiselyov and C.-c. Shan. "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages". In: Journal of Functional Program- ming 19.5 (Sept. 2009), pp. 509–543.

[5] J. Cheney and R. Hinze. First-class phantom types. Tech. rep. Cornell University, 2003.

[6] J. van Diggelen, W. Post, M. Rakhorst, R. Plasmeijer and W. van Staal. "Using Process- Oriented Interfaces for Solving the Automation Paradox in Highly Automated Navy Ves- sels". In: Active Media Technology. Ed. by D. S´le¸zak, G. Schaefer, S. T. Vuong and Y.-S. Kim. Cham, Switzerland: Springer International Publishing, 2014, pp. 442–452.

[7] L. Domoszlai, B. Lijnse and R. Plasmeijer. "Parametric Lenses: Change Notification for Bidirectional Lenses". In: Proceedings of the 26nd 2014 International Symposium on Im- plementation and Application of Functional Languages. IFL '14. ACM, 2014, 9:1–9:11.

[8] T. Elliott et al. "Guilt Free Ivory". In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell. Haskell '15. Vancouver, BC, Canada: ACM, 2015, pp. 189–200. isbn: 978-1- 4503-3808-0.

[9] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce and A. Schmitt. "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem". In: ACM Trans. Program. Lang. Syst. 29.3 (May 2007).

[10] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. 2017. url: https://www.gartner.com/en/newsroom/press-releases/2017-02- 07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31- percent-from-2016 (visited on 04/08/2018).

[11] M. Grebe and A. Gill. "Haskino: A Remote Monad for Programming the Arduino". In: Practical Aspects of Declarative Languages. Ed. by M. Gavanelli and J. Reppy. Cham, Switzerland: Springer International Publishing, 2016, pp. 153–168.

[12] P. C. Hickey, L. Pike, T. Elliott, J. Bielman and J. Launchbury. "Building Embedded Systems with Embedded DSLs". In: SIGPLAN Not. 49.9 (Aug. 2014), pp. 3–9.