Rapid prototyping of IoT applications with Esperanto compiler

Gyeongmin Lee
POSTECH
Pohang, Republic of Korea
paina@postech.ac.kr

Seonyeong Heo
POSTECH
Pohang, Republic of Korea
heosy@postech.ac.kr

Bongjun Kim
POSTECH
Pohang, Republic of Korea
bong90@postech.ac.kr

Jong Kim
POSTECH
Pohang, Republic of Korea
jkim@postech.ac.kr

Hanjun Kim
POSTECH
Pohang, Republic of Korea
hanjun@postech.ac.kr

ABSTRACT

Integrating various networked devices, the Internet of Things (IoT) enables various new services like home automation, making its market larger and more competitive. Although rapid development of an IoT application is crucial to keep up with the highly competitive IoT market, developing an IoT application is challenging for programmers because the programmers should integrate multiple programmable devices and heterogeneous third-party devices. Some IoT frameworks integrate programming environments of multiple devices, but they either require device-specific implementation for third-party devices without any device abstraction, or abstract all the devices to the standard interfaces requiring unnecessary abstraction of programmable devices. This work introduces the Esperanto framework that integrates IoT devices with selective abstraction, allowing rapid prototyping of an IoT application. Exploiting the correspondence between an object and a thing in the object oriented programming (OOP) model, the Esperanto framework allows programmers to write only one OOP program instead of multiple programs for each device, and to manipulate third-party devices with their common ancestor classes. Compared to an existing approach on the integrated IoT programming, Esperanto requires 33.3% fewer lines of code to implement 5 IoT services, and reduces their response time by 44.8% on average. Moreover, with an empirical study, this work shows that the Esperanto framework reduces the development time by 52.7%.

CCS CONCEPTS

• **Hardware** → **Emerging languages and compilers**; • **Software and its engineering** → *Distributed programming languages*;

RSP'17, October 15–20, 2017, Seoul, Republic of Korea

ACM ISBN 978-1-4503-5418-9/17/10…$15.00
https://doi.org/10.1145/3130265.3138857

KEYWORDS

Internet of Things, IoT, Rapid prototyping, Esperanto

ACM Reference Format:
Gyeongmin Lee, Seonyeong Heo, Bongjun Kim, Jong Kim, and Hanjun Kim. 2017. Rapid prototyping of IoT applications with Esperanto compiler. In *Proceedings of RSP'17, Seoul, Republic of Korea, October 15–20, 2017,* 7 pages.
https://doi.org/10.1145/3130265.3138857

1 INTRODUCTION

Integrating various networked devices, the Internet of Things (IoT) enables new services such as home automation and health monitoring. Figure 1 shows a baby monitor application as an IoT service example. In the example, if the baby cries, an IP camera notifies parents by blinking smartbulbs and sending a message to mobile phones. Like this example, the IoT environment enables new promising services, and the IoT market will be larger and more competitive as more and more devices are connected to the Internet.

Although rapid development of an application is crucial for IoT application programmers to keep up with the highly competitive IoT market, building an IoT application is challenging and time-consuming due to multiple programmable devices and heterogeneous APIs of third-party devices. To integrate multiple programmable IoT devices, programmers should write multiple disjoint sub-programs for each device, and explicitly manage communication among the devices. Moreover, since different vendors adopt different APIs for their devices, programmers should add device-specific implementation for similar third-party devices like Hue and LIFX smartbulbs. Thus, to simplify IoT programming, an IoT programming platform should integrate multiple programmable IoT devices while abstracting various APIs of similar third-party devices into common APIs.

Though recent works [1, 4–6, 9–11, 14–17] have proposed various IoT platforms that reduce the development time, but none of them can fully solve the challenges. Protocol integration platforms [4, 9, 17] unify communication protocols across IoT devices, but they still require programmers to write and synchronize multiple sub-programs without a holistic view of an IoT application. Device integration platforms [1, 6, 14, 15, 18] integrates multiple IoT devices with a holistic view of an application. However, they either require

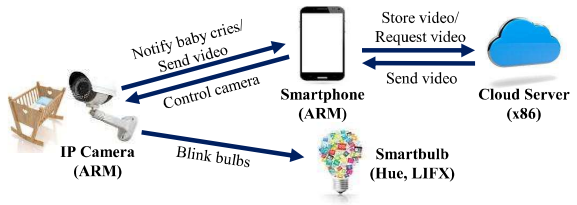

Figure 1: An IoT service example: Baby monitor

Syntax of the Esperanto primitives	
Device Declaration	`#pragma EspDevDecl(devID,main)`
Device-Object Mapping	`#pragma EspDevice(devID,(conditions))` `class className;`
3rd-party Device Import	`#pragma EspImport(className,funcName)`
Runtime variable examples in `condition`	
`TYPE`	Device type (e.g. Server, Mobile, Bulb)
`VENDOR`	Vendor of device
`MODEL`	Model name of device
`ARCH`	Processor architecture of device
`OS`	Operating system of device

Table 1: Esperanto syntax

device-specific implementation for third-party devices without abstracting heterogeneous APIs into a common API [1, 6, 14], or require unnecessary abstraction of programmable devices to the standard interfaces considering all the devices as third-party devices [15, 18].

This work introduces rapid prototyping of IoT applications with the Esperanto compiler-runtime framework [12]. Exploiting the correspondence between an object and a thing in the object oriented programming (OOP) model, The Esperanto language allows programmers to write only one OOP program instead of multiple programs for each device, and to manipulate third-party devices with their common ancestor classes. The Esperanto compiler automatically partitions the integrated OOP program into multiple sub-programs for each IoT device, and inserts communication and synchronization code. The Esperanto runtime dynamically binds the ancestor class to its descendant objects reflecting connected third-party devices at run-time.

To evaluate the Esperanto framework for rapid prototyping, this work implements 14 events of 5 IoT services such as baby monitor, fitness tracking, taxi application, heart attack detection, and fire alarm application in the Esperanto language and an existing device integration approach. The Esperanto framework requires 33.3% fewer lines of code with 44.8% shorter response time on average. Moreover, this work executes an empirical study that shows that the Esperanto language reduces the development time by 52.7%.

2 ESPERANTO FRAMEWORK

To effectively reduce the burden of IoT programmers, the proposed IoT framework should be simple and easy for programmers to learn and use, and also be powerful enough to solve the challenges such as device integration and abstraction. The Esperanto framework [12] revisits the object oriented programming (OOP) model that most programmers are familiar with. In the OOP model, objects correspond to things in the real world, and a parent class abstracts its children classes. With a minimal extension of an existing OOP language, three new annotations in total, the Esperanto framework integrates multiple programmable devices with one OOP program, and selectively abstracts heterogeneous APIs of third-party devices to their parent classes.

Integrated programming: An IoT service is composed of things and communications among things. Similarly, an OOP program is composed of objects and describes interactions between objects. Based on the correspondence between objects and things, the Esperanto language integrates multiple sub-programs of IoT devices in an IoT service into a single integrated OOP program. As an OOP programmer manages multiple objects in a single OOP program, an Esperanto programmer manages multiple IoT devices with a single Esperanto program. According to the Esperanto primitives, the Esperanto compiler partitions the integrated program into multiple sub-programs, and inserts communication codes for method calls at different objects.

Device abstraction: The concept of inheritance and polymorphism gives "is-a" relationships between objects and allows polymorphic behaviors while providing a common interface to objects. Exploiting this concept, the Esperanto language abstracts similar types of devices to have a common interface and binds device-specific implementation of the devices to the interface. For the abstract interface, the Esperanto compiler dynamically links its corresponding object implementation reflecting the execution environment.

2.1 Esperanto Language

This work introduces the Esperanto language [12] that extends the existing C++ language with three annotations for the easy and powerful integrated IoT programming with selective abstraction. Though the Esperanto language extends the C++ language, the proposed syntax and semantics are not tied to C++ because the proposed language does not require any C++ specific feature. Table 1 and Figure 2 show the three annotations and the Esperanto codes for the baby monitor example in Figure 1.

`EspDevDecl` declares a programmable device with its name (`devID`), constructor and destructor. The constructor and the destructor will be invoked at the beginning and the end of the sub-program of the device. For example, Figure 2 declares two programmable devices such as `Cam` (IP Camera) and `Phone` with their constructors and destructors (Lines 3-4). Here, `EspDevDecl` is only allowed for programmable devices because the Esperanto compiler generates sub-program binaries for the `EspDevDecl` annotated devices.

`EspDevice` maps its annotated class to the declared device. For example, the Esperanto programmer can install the `IPCamera` class at the device `Cam` with the `EspDevice` annotation at line 10 in Figure 2. Here, the programmer inserts the annotation only for code that should be executed in the device. Given the annotation, the Esperanto compiler will

```
1  /* *** BabyMonitor.h/cpp *** */
2  // Declare all the programmable devices
3  #pragma EspDevDecl(Cam, cam_ctor, cam_dtor)
4  #pragma EspDevDecl(Phone, m_ctor, m_dtor)
5
6  // Generate an import function for 3rd-party devices
7  #pragma EspImport(SmartBulb, getBulbs)
8
9  // Map IPCamera class to device Cam
10 #pragma EspDevice(Cam)
11 class IPCamera {
12 private:
13   void onBabyCry();
14 };
15
16 // Map Mobile class to device Phone
17 #pragma EspDevice(Phone)
18 class Mobile {
19 public:
20   void alarm(string msg);
21 };
22
23 IPCamera* cam;
24 List<Mobile*> m_list;
25 SmartBulb** bulbs;
26 int num_bulbs = 0;
27
28 // Work as main function of Cam device
29 void cam_ctor(){
30   cam = new IPCamera();
31   // Bind all the SmartBulb instances
32   bulbs = getBulbs(&num_bulbs);
33 }
34
35 void IPCamera::onBabyCry(){
36   for(size_t i=0;i<m_list.size();i++){
37     // Send a message through a function call
38     m_list[i]->alarm("Baby is crying");
39   }
40   // Device abstraction for SmartBulbs
41   for(size_t i=0;i<num_bulbs;i++) bulbs[i]->blink();
42 }
43
44 // A mobile phone registers itself to a m_list
45 void m_ctor(){
46   Mobile* m = new Mobile();
47   m_list.push_back(m);
48 }
```

Figure 2: Esperanto pseudo codes of the baby monitor application in Figure 1

automatically map non-annotated instructions to appropriate IoT devices based on performance estimation results.

`EspDevice` can optionally pass device conditions as an argument to specify its target physical device. The Esperanto compiler framework requires hardware description of each device such as its device type, vendor, model, architecture and operating system. The Esperanto runtime dynamically checks the description, and maps the object to the appropriate device. For example, Figure 3 shows that `Bulb` device programmers annotate their classes with `VENDOR` and `MODEL` runtime variables (Line 2 in Hue.h and 2 and 9 in LIFX.h). According to the annotated condition, the runtime maps `Hue`, `LIFX`, and `LIFXZ` classes to their corresponding physical devices such as Hue, LIFX and LIFXZ smartbulbs.

`EspImport` and its import function allow IoT application programs to exploit non-programmable third-party devices like Hue and LIFX bulbs. `EspImport` generates an import

```
1  /* *** SmartBulb.h *** */
2  #pragma EspDevice(Bulb, TYPE==BULB)
3  class SmartBulb {
4  public:
5    virtual bool connect() = 0;
6    virtual void blink() = 0;
7  };
```

```
1  /* *** Hue.h *** */
2  #pragma EspDevice(Bulb, VENDOR==PHILIPS)
3  class Hue : SmartBulb {
4  public:
5    bool connect();
6    void blink();
7  private:
8    char bridge[23];
9  };
```

```
1  /* *** LIFX.h *** */
2  #pragma EspDevice(Bulb, VENDOR==LIFX)
3  class LIFX : SmartBulb {
4  public:
5    bool connect();
6    void blink();
7  };
8
9  #pragma EspDevice(Bulb, MODEL==LIFXZ)
10 class LIFXZ : LIFX{
11 public:
12   void blink(int idx);
13 };
```

Figure 3: SmartBulb class and its descendant classes

function that binds all the `className` devices (Line 7), and the import function returns `className` objects of all the connected devices (Line 32). Programmers can specify a certain type of devices by passing its corresponding class type as the `className` argument. For example, if a programmer uses `LIFX` instead of `SmartBulb` as the first argument, the import function returns connected `LIFX` and `LIFXZ` objects but does not return `Hue` objects.

`Runtime variables` are hardware and system information of connected devices such as their device type, vendor, architecture and operating system. With the runtime variables, programmers can specify a target device that an object is mapped on.

2.2 Esperanto Compiler

The Esperanto compiler [12] transforms an integrated Esperanto program into multiple sub-programs for IoT devices. First, the compiler marks all the instructions in a program with their target devices. Then, the compiler divides the marked instructions into multiple sub-programs, and inserts communication instructions. Finally, the compiler customizes each sub-program for each device with back-end compilers.

To mark instructions in a program, the compiler recognizes the Esperanto syntax, and maps all the instructions into their target devices. Figure 4 shows how the compiler transforms the Esperanto program with code snippets. The Esperanto parser recognizes the Esperanto syntax, and changes each pragma to a metadata. The metadata generator annotates all the instructions in a device-annotated class with their target devices. For example, the parser and the metadata

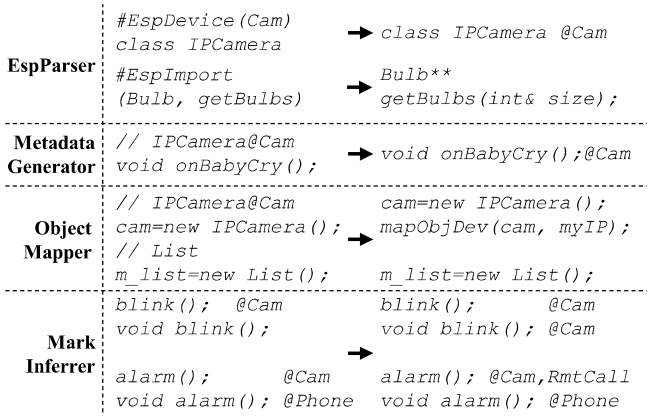

Figure 4: Marking Process

generator mark `IPCamera` class and its member functions as `Cam` device. The object mapper inserts a call instruction of `mapObjDev` where a `EspDevDecl`-annotated object is allocated. The `mapObjDev` function registers the pointer of the newly allocated memory object and the current network IP into the object-device map in the runtime. The mark inferrer marks all the non-annotated functions. The mark inferrer estimates performance for possible devices, and annotates the optimal device to the functions. If the caller and callee instructions are marked as different devices, the mark inferrer annotates the caller as a remote function call (`RmtCall`) that requires network communication.

After marking instructions, the compiler automatically partitions the Esperanto program into multiple sub-programs for each IoT device, and inserts communication instructions as Figure 5 illustrates. First, the compiler replaces all the remote function call sites marked as `RmtCall` with the network communication `send` function calls. The compiler passes the memory address of callee objects as the first argument of the `send` function. Since `mapObjDev` registers the object address with its IP, the `send` function can find the target IP from the object address (Figure 6). Then, the compiler generates multiple sub-programs and erases instructions that are not marked in each sub-program. Finally, the compiler unifies heterogeneous memory layouts such as alignment, pointer size and endianness across different devices.

For `EspImport` to bind devices in a resource-centric and device agnostic way, the Esperanto compiler framework automatically generates a SW description file that includes resources of each object, and an import function that allocates device objects reflecting underlying physical devices. The compiler analyzes each `EspDevice` annotated class with the annotated conditions, and writes its class hierarchy and public `EspDevice` member variables to the SW description file. Based on the analyzed class hierarchy, the compiler generates an import function for each `EspDevice` annotated class. The import function dynamically reads HW descriptions of connected devices, compares the descriptions with conditions

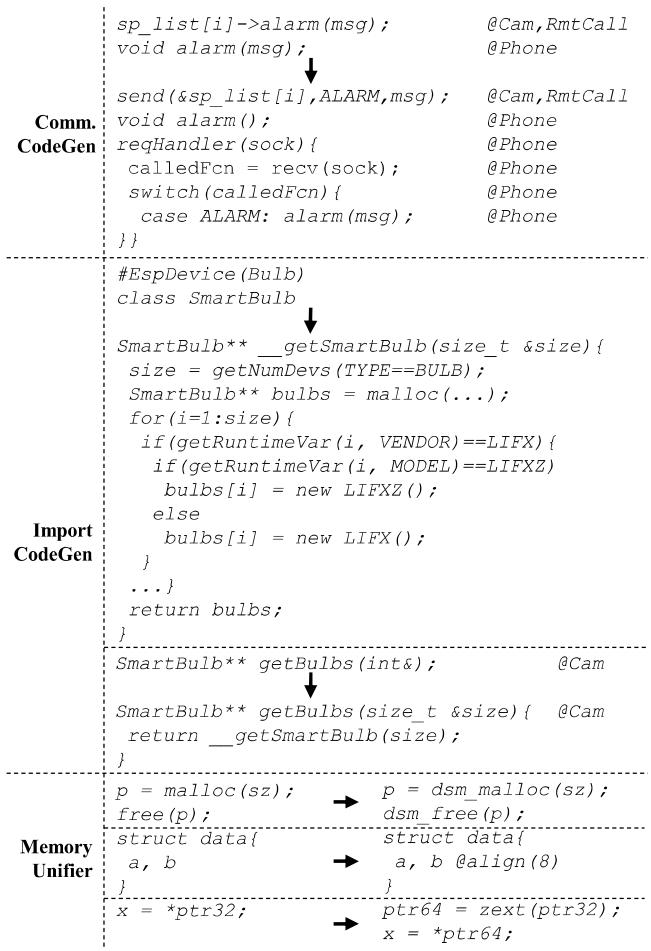

Figure 5: Partitioning Process

of the annotated class and its descendant classes, and allocates corresponding objects. Figure 5 shows how the import code generator transforms `EspDevice` and `EspImport` annotations into an import function and its wrapper function.

2.3 Esperanto Runtime

The Esperanto runtime [12] manages all the connected IoT devices at the user environments, and supports multiple programmable device integration and selective abstraction. The runtime consists of three modules such as a device manager, a communication manager, and a memory manager.

The device manager manages all the connected IoT devices and supports abstraction of third-party devices. When a new IoT device is connected to the user environment, the device manager collects HW description about the device such as IP address, device type, vendor, architecture, and operating system (Steps 1 to 3 in Figure 6). Here, the Esperanto runtime periodically checks connected IoT devices to find the third-party IoT devices that do not include the Esperanto runtime. When an Esperanto program searches devices with a runtime

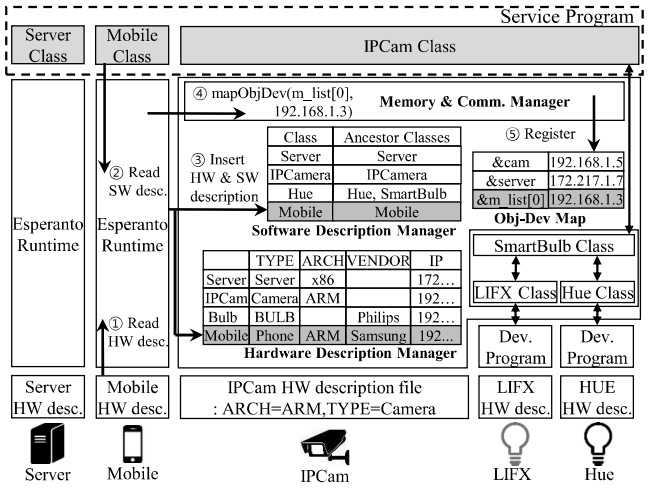

Figure 6: The overall structure of the runtime

condition such as `TYPE==BULB`, the device manager searches corresponding devices from the HW description.

To support device abstraction, the device manager also keeps SW description that includes all the ancestor classes of a connected device. If an Esperanto program imports one of the ancestor classes using `EspImport`, the runtime checks the SW description and returns the connected devices. Then, the Esperanto program creates appropriate objects for the devices with their HW description. For example, the `__getSmartBulbs` function in Figure 5 requests the Esperanto runtime to search third-party devices that have `SmartBulb` as its ancestor class. The Esperanto runtime finds descendent classes of `SmartBulb` from the SW description and returns their connected device lists from the HW description such as `Hue`, `LIFX` and `LIFXZ`. As Figure 5 illustrates, the `__getSmartBulb` function creates and returns corresponding descendent objects for the devices in the lists.

The communication manager maps objects in an Esperanto program to a physical device, and manages communication among objects. The compiler inserts an object-device mapping function call (`mapObjDev`) for every `EspDevice`-annotated object allocation. When `mapObjDev` is invoked, the communication manager inserts a new element to the object-device map. Steps 4 and 5 in Figure 6 show how the Esperanto runtime maps a newly created device object to a connected physical device. If `send` is invoked, the device manager finds a corresponding IP address with the memory address of the passed-in object from the object-device map, and sends the message to the target IP address.

The memory manager in the Esperanto runtime manages memory coherence across IoT devices at run-time. Since the memory unifier in the Esperanto compiler unifies heterogeneous memory layouts across heterogeneous IoT devices as one layout, the memory manager in the runtime only needs to support memory coherence without worrying about memory translation.

Service	Event	Description
`Baby Monitor`	Alarm	Notifies the baby cry to bulb and mobile
	Cam	Send an image frame to mobiles and upload the frame to the server
`Fitness Tracking`	Register	Register a mobile device into the server
	Scale	Send weight and body fat to mobiles
	Update	Update tracking and weight history
	Report	Generate an analysis report
`Taxi App.`	Register	Register a taxi driver into the server
	Loc	Update the location of a taxi
	Call	Call a nearby taxi
`Heart Attack`	Register	Register a hospital server into the server
	Alarm	Notify heart attack to a nearby hospital
	SendRate	Send heart rates to history
`Fire Alarm`	Info	Send temperatures to the server
	Alarm	Notify a fire alarm to mobiles

Table 2: Evaluated event description

Service	Device	Specification
`Baby Monitor`	IP Camera	ODROID-XU4 with USB-CAM 720P (Samsung Exynos 5422, 2GB)
	Mobile	Samsung Galaxy S5 (Qualcomm Snapdragon 801, 3GB)
	Server	Desktop Server (Intel Core i7-6700, 16GB)
	Bulb	Philips Hue and LIFX
`Fitness Tracking`	Server	Desktop Server
	Mobile	Samsung Galaxy S5
	SmartScale	ODROID-C0 (Amlogic ARM Cortex-A5, 1GB)
	SmartBand	Gear Fit and Mi Band
`Taxi App.`	Server	Desktop Server
	Driver	ODROID-XU4
	Customer	Samsung Galaxy S5
`Heart Attack`	Gateway	Desktop Server
	Mobile	Samsung Galaxy S5
	Hospital	Desktop Server
	SmartBand	Gear Fit and Mi Band
`Fire Alarm`	Server	Desktop Server
	Mobile	Samsung Galaxy S5
	Thermometers	ODROID-XU4 with a weather board
	Bulb	Philips Hue and LIFX

Table 3: IoT Service and device specification

3 EVALUATION

To evaluate the Esperanto language and framework, this work implements 5 IoT services such as baby monitor, fitness tracking, taxi application, heart attack detection and fire alarm application, which support 14 events in total. Table 2 briefly describes each event. Moreover, this work deploys the services on heterogeneous IoT devices ranging from embedded systems such as ODROID-XU4 and ODROID-C0 [7] to a Samsung Galaxy S5 mobile phone and a desktop server. The mobile runs the Android 4.4.2 (KitKat), the desktop server runs Ubuntu 16.04, and the embedded systems run Ubuntu MATE 1.10.2. Here, to evaluate programmability on a device-specific custom hardware, this work installs a camera hardware module and a weather board on the embedded systems. Third-party devices such as smartbulbs (Hue and LIFX) and smart bands (Gear Fit and Mi Band) are used to evaluate the third-party device management in a device agnostic way. All the ODROID devices are wirelessly connected with 144Mbps maximum bandwidth (802.11n), and

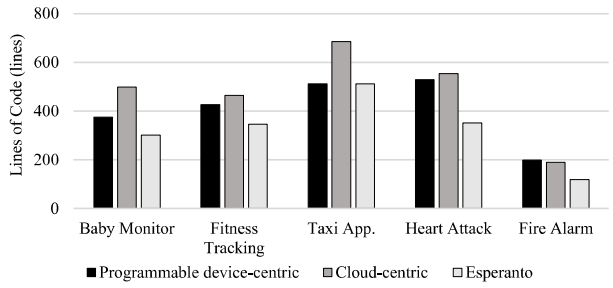

Figure 7: Lines of code of the IoT programs

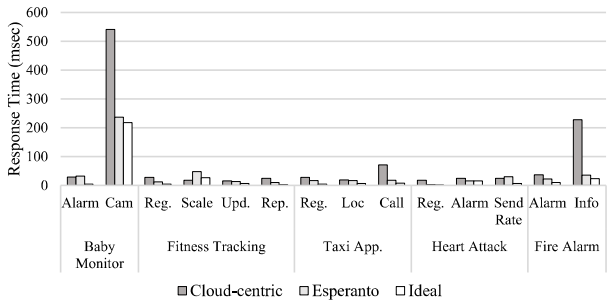

Figure 8: Response time of each event

the mobile is connected with 844Mbps (802.11ac). Table 3 describes devices used in each service.

3.1 Programmability of Esperanto

To compare the programmability of the Esperanto language with other IoT programming models, this work also implements the same services in programmable device-centric and cloud-centric integration approaches. The programmable device-centric approach integrates multiple programmable devices into one programming environment like Esperanto, but it does not support any device abstraction requiring device-specific implementation for each third-party devices. The cloud-centric integration approach integrates IoT devices with standard abstraction, so it requires unnecessary device handlers and device programs for programmable devices. Here, to eliminate performance effects from the underlying platforms, all the services are written in C++.

Figure 7 shows that Esperanto successfully simplifies IoT programming. For the same 5 IoT services, the average lines of the Esperanto programs are 23.8% and 33.3% shorter than lines of programmable device-centric and cloud-centric programs. The lines of code consist of third-party device management, device handlers and service algorithm. All the programs do not include any device registration, identification and explicit communication code among devices because the three approaches support integrated IoT programming with a holistic programming view. Since Esperanto supports device agnosticism without requiring device-customized code for third-party devices, the Esperanto programs have fewer lines of code than the programmable device-centric ones.

Questions	Device-centric	Esperanto
G1 Average LoC	302.6	118
G1 Average development time (mins)	401	151.4
G2 Average LoC	310.4	128.4
G2 Average development time (mins)	307.6	183.6
Average LoC	306.5	123.2
Average development time (mins)	354.3	167.5
Ease of learning (1:Difficult, 5:Easy)	3.2	4.3
Ease of use (1:Difficult, 5:Easy)	2.5	4.6
Ease of debugging (1:Difficult, 5:Easy)	2.6	3.2
Applicability for IoT programming (1:Not Applicable, 5:Applicable)	2.7	4.2

Table 4: Empirical study results

Here, if the programs support more third-party devices, the programmable device-centric programs will have more lines of code because the lines of device-customized code is proportional to the number of third-party devices. Moreover, the Esperanto programs do not include device handlers for programmable devices, so the Esperanto programs have also fewer lines of code than the cloud-centric ones. As a result, Esperanto requires only service algorithm codes and effectively reduces the burden of programmers.

3.2 Response Time Analysis

This section evaluates performance of the cloud-centric integration approach and Esperanto. This work does not evaluate the programmable device-centric programs because the programmable device-centric programs and the Esperanto program have the same communication topology. Instead of the programmable device-centric programs, this work manually implements the ideal IoT services that do not include any platform overheads with optimal communication to analyze the maximum achievable performance of the services.

Figure 8 shows the response time of each event in the IoT programs. The results are the average response time of ten invocations per event. Since Esperanto allows IoT devices to directly communicate with each other without passing through the cloud server, Esperanto shows average 44.8% shorter response time than cloud-centric integration approach. Compared to the ideal programs, the Esperanto programs suffer from average 12.56 milliseconds and up to 28.14 milliseconds (`Baby Monitor Alarm`) latency overheads that are negligible enough for people not to recognize [13].

3.3 Empirical Study

To deeply evaluate programmability of the Esperanto language, this work executes an empirical study with 10 junior and senior computer science undergraduate students who have experiences in network programming. In the empirical test, all the participants learn the device-centric and Esperanto programming models, and develop one event (`Cam`) of the baby monitor program in each model. To remove side-effects from the order of the programming models, this work divides the participants into two groups. Participants in group 1 develop the program in the device-centric model,

and then develop the program in the Esperanto language. Participants in group 2 develop the program in the opposite order.

Table 4 shows the results of the empirical study. The Esperanto language allows the participants to write 2.48 times shorter programs and reduces their development time by 52.7%, compared to the device-centric programming model. The survey results also show that the participants easily learn and use the Esperanto language. The participants answer that debugging is a little difficult in the Esperanto language due to unfriendly compile error message. Generating detailed compile error messages and debugging tools will make the debugging easier, and this work leaves it as a future work.

4 RELATED WORK

To reduce the burden of IoT programmers, previous works [2, 3, 5, 6, 8, 10, 11, 14–16, 19] have proposed integrated programming models and platforms for distributed systems including IoT and wireless sensor networks. Like Esperanto, the programming models and platforms integrate programming environments of heterogeneous devices and provide a holistic view of an application to programmers.

SmartThings [15] is a programming platform for IoT that encapsulates a physical device as a composition of its capabilities. The SmartThings capability model allows an application programmer to write an IoT service program in a device agnostic way by providing device management in capability granularity. However, since the framework only supports standard capabilities, it limits its applicability for custom programmable devices. Unlike SmartThings, Esperanto allows the programmer to directly manage programmable devices without the standardization of the devices.

Node-RED [14] and its extensions such as distributed Node-RED [6] and glue.things [11] provide a graphical user interface for integrated IoT programming. With the proposed systems, programmers can design an IoT system by graphically combining devices and specifying data flows. However, due to their lack of support for third-party integration in a device agnostic way, the programmers should implement different binding codes to control various third-party devices. On the other hand, this work supports device agnosticism by exploiting the inheritance and polymorphism features of the object oriented programming model.

nesC [5], Eon [16] and Mace [10] are extended C/C++ languages that support a holistic design of networked embedded systems. These languages provide abstraction with high level objects and connect components with their interfaces. Although they simplify complex event handling implementation, the languages have little consideration for device agnosticism since they do not target IoT environments.

5 CONCLUSION

This work introduces the Esperanto framework that integrates IoT devices with selective abstraction, and shows that the Esperanto framework successfully enables rapid prototyping of an IoT application. Compared to an existing approach on the integrated IoT programming, Esperanto requires 33.3% fewer lines of code to implement 5 IoT services, and reduces their response time by 44.8% on average. With an empirical study, this work also shows that the Esperanto framework reduces the development time by 52.7%.

ACKNOWLEDGMENTS

We thank the CoreLab and HPC Lab for their support and feedback during this work. This work is supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1403-04 and SRFC-TB1703-03.

REFERENCES

[1] Michael Blackstock and Rodger Lea. 2014. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *Proceedings of the 5th International Workshop on Web of Things*.
[2] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*.
[3] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*.
[4] Eclipse SmartHome 2017. http://eclipse.org/smarthome. (2017).
[5] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[6] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. 2015. Developing IoT applications in the fog: a distributed dataflow approach. In *Proceedings of International Conference on the Internet of Things*.
[7] Hardkernel:ODROID 2017. http://www.hardkernel.com. (2017).
[8] Galen C. Hunt and Michael L. Scott. 1999. The Coign Automatic Distributed Partitioning System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*.
[9] IoTivity Project 2017. https://www.iotivity.org. (2017).
[10] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[11] Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. 2014. Glue.Things: A Mashup Platform for Wiring the Internet of Things with the Internet of Services. In *Proceedings of the 5th International Workshop on Web of Things*.
[12] Gyeongmin Lee, Seonyeong Heo, Bongjun Kim, Jong Kim, and Hanjun Kim. 2017. Integrated IoT Programming with Selective Abstraction. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*.
[13] Robert B. Miller. 1968. Response Time in Man-computer Conversational Transactions. In *Proceedings of AFIPS Fall Joint Computer Conference*.
[14] Node-RED 2017. http://nodered.org. (2017).
[15] SmartThings 2017. http://www.smartthings.com. (2017).
[16] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*.
[17] The Thing System 2017. http://thethingsystem.com. (2017).
[18] xively by LogMein 2017. http://www.xively.com. (2017).
[19] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. 2014. Customizable and Extensible Deployment for Mobile/Cloud Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation*.